SxPipe 2: architecture pour le traitement pré-syntaxique de corpus bruts

Benoît Sagot, Pierre Boullier

SXPipe 2: architecture pour le traitement pré-syntaxique de corpus bruts

Benoît Sagot — Pierre Boullier

INRIA Paris-Rocquencourt - Projet ALPAGE
Domaine de Voluceau
Rocquencourt, B.P. 105
F-78153 Le Chesnay cedex
{benoit.sagot,pierre.boullier}@inria.fr

RÉSUMÉ. *Cet article présente* SXPipe *2, chaîne modulaire et paramétrable dont le rôle est d'appliquer à des corpus bruts une cascade de traitements de surface. Préalable nécessaire à une possible analyse syntaxique, ils peuvent également servir à préparer d'autres tâches. Développé pour le français mais également pour d'autres langues,* SXPipe *2 comprend, entre autres, divers modules de reconnaissances d'entités nommées dans du texte brut, un segmenteur en phrases et en tokens, un correcteur orthographique et reconnaisseur de mots composés, ainsi qu'une architecture originale de reconnaissance de motifs non contextuels, utilisée par différentes grammaires spécialisées (nombres, constructions impersonnelles,...). Nous présentons les fondements théoriques des différents modules, leur mise en œuvre pour le français et pour certains une évaluation quantitative.*

ABSTRACT. *This article introduces* SXPipe *2, a modular and customizable chain aimed to apply to raw corpora a cascade of surface processing steps. Necessary preliminary step before parsing, they can be also used to prepare other tasks. Developed for French and for other languages,* SXPipe *2 includes, among others, various named entities recognition modules in raw text, a sentence segmenter and tokenizer, a spelling corrector and compound words recognizer, and an original context-free patterns recognizer, used by several specialized grammars (numbers, impersonal constructions,...). We describe the theoretical foundations of these modules, their implementation on French and a quantitative evaluation for some of them.*

MOTS-CLÉS : *Reconnaissance d'entités nommées, correction orthographique, mots composés, tokenisation, segmentation, DAG*

KEYWORDS: *Named entities recognition, spelling correction, compound words, tokenization, sentence segmentation, DAG*

1. Introduction

Dans le domaine du Traitement Automatique des Langues (TAL), des tâches aussi importantes que l'analyse syntaxique, l'extraction ou la recherche d'informations ont a traiter des corpus bruts. Ces corpus sont souvent bien différents de ce qu'attendent les systèmes automatiques, à cause d'un large éventail d'imperfections, qui nécessitent des traitements adaptés, appelés pré-traitements ou traitements de surface. La plupart de ces traitements peuvent être classées en trois catégories : détection de frontières (entre phrases, entre mots), correction d'erreurs (correction orthographique, bruit typographique) et « entités nommées » (séquences de caractères ou de mots absents d'un lexique de la langue mais issus de mécanismes productifs, comme les adresses, les dates, les acronymes, et bien d'autres[1]). De plus, certains traitements de surface peuvent guider utilement l'analyse syntaxique.

Les pré-traitements sont parfois considérés comme des tâches aisées, sur lesquelles il n'est ni utile ni intéressant de se pencher, alors qu'ils jouent un rôle crucial dans la qualité des traitements ultérieurs. De nombreux outils existent déjà, parmi lesquels nous évoquerons Unitex (Paumier, 2003), GATE (Cunningham *et al.*, 2002) et LinguaStream (Bilhaut et Widlöcher, 2006). Cependant, l'expérience montre que les outils disponibles ne sont pas toujours satisfaisants pour une application donnée, soit qu'ils ne disposent pas d'un module de correction orthographique, soit qu'ils soient spécialisés pour un certain type de corpus, soit qu'ils ne sachent pas gérer de façon satisfaisante les ambiguïtés qui peuvent apparaître à chaque étape.

C'est pour répondre à ces insuffisances que nous avons développé une nouvelle architecture modulaire et adaptative pour le pré-traitement de corpus brut, nommée SXPipe. Elle permet de transformer un texte brut en un DAG[2], voir ci-dessous) de formes, entrée valide pour un analyseur syntaxique complet, pour un outil de normalisation textuelle (par re-transformation en texte brut), ou pour un système d'extraction d'informations *via* des entités nommées. SXPipe 2 permet le traitement de corpus variés, intègre un module non-déterministe de correction orthographique et d'identification des formes composées, et permet la reconnaissance de divers types d'entités nommées et d'autres types de motifs. Enfin, SXPipe gère les ambiguïtés de façon satisfaisante.

Cet article présente la deuxième version de SXPipe. Depuis la première version, décrite par exemple dans (Sagot et Boullier, 2005) et (Boullier *et al.*, 2005), la chaîne a évolué de façon considérable :

– SXPipe 2 est une chaîne multilingue (gestion du multilinguisme, chaîne opérationnelle pour le polonais, chaînes en cours de perfectionnement pour l'anglais, l'espagnol, le slovaque et le slovène) ;

– SXPipe 2 dispose d'un module unifié de correction orthographique et de recon-

1. Nous utilisons donc ce terme dans un sens légèrement plus large que celui couramment accepté, cf. section 5.
2. *Directed Acyclic Graph* (graphe orienté acyclique).

naissance de formes composées, qui peut produire des sorties ambiguës : la correction orthographique, lieu d'incertitudes aussi important que la reconnaissance des mots composés, peut maintenant se faire de façon non-déterministe ;

– SXPipe 2 intègre une architecture originale pour la reconnaissance de motifs décrits à l'aide de grammaires non contextuelles (CFG) ;

– de nombreuses améliorations ont été apportées aux divers composants de la chaîne, qu'il s'agisse des modules de reconnaissance d'entités nommées ou, plus récemment, de la compilation efficace de lexiques au moyen de structures compactes, rapides à produire et très rapides à parcourir.

L'intégralité de SXPipe 2 est distribué sous licence Cecill-C (compatible LGPL), en tant que module de la chaîne de traitement de l'équipe Alpage[3].

La section 2 rappelle les caractéristiques principales des trois autres architectures de traitement citées ci-dessus, à savoir Unitex, GATE et LinguaStream, en montrant les spécificités, les avantages (et les points faibles) de SXPipe. La section 3 constitue des préliminaires nécessaires à la description de SXPipe 2 : nous définissons les termes et les notions que nous utiliserons tout au long de cette article, ainsi que le format de représentation manipulé par SXPipe. La section 4 présente l'architecture de SXPipe de façon détaillée et motivée, en mettant l'accent sur le rapport entre traitement séquentiel et gestion des ambiguïtés.

Les sections 5 à 7 présentent en détails les différents composants, en les répartissant en trois ensembles : la section 5 décrit les composants agissant au niveau du texte puis le découpant en tokens ; la section 6 décrit les composants agissant au niveau des tokens puis les corrigeant et/ou les re-répartissant de façon ambiguë en un graphe de formes (plus exactement, un DAG ; la section 7 décrit les composants agissant au niveau des formes, composants qui reposent sur le module original de reconnaissance (non-déterministe) de motifs non contextuels.

L'évaluation d'une chaîne comme SXPipe est délicate pour au moins deux raisons. Tout d'abord, il n'existe aucun corpus de référence associant à du texte brut la séquence des formes qui le composent. Ensuite, SXPipe est modulaire et paramétrable, en ce sens qu'il s'agit d'un ensemble de composants qui sont souvent paramétrables : changer les paramètres d'un composant tel que le correcteur orthographique peut conduire à des résultats très différents (selon certains seuils, un mot peut ainsi recevoir une correction ou être identifié comme un mot inconnu). Mais cette adaptabilité est un point fort, car on peut adapter la chaîne, statiquement ou dynamiquement, au type de corpus à traiter. Nous avons malgré tout réalisé un certain nombre d'expériences permettant de se faire une idée de la qualité de SXPipe 2, dans sa configuration par défaut pour le français. Ces évaluations sont réparties au fil de l'article, en complément à la description des modules concernés. Avant de conclure et d'indiquer

3. Projet `lingwb` sur INRIAGForge : `http://gforge.inria.fr/projects/lingwb/`. Site internet de `lingwb` : `http://lingwb.gforge.inria.fr/`.

quelques perspectives (section 9), nous fournissons quelques données sur l'application de SXPipe à corpus journalistique de plus de 17 millions de mots (section 8).

2. Comparaison avec d'autres chaînes de traitement

SXPipe n'est pas, loin s'en faut, la première chaîne de traitement pour le français, ou compatible avec des outils traitant le français. Parmi les systèmes les plus couramment utilisés, nous allons rapidement passer en revue trois plateformes : Unitex (Paumier, 2003), GATE (Cunningham *et al.*, 2002) et LinguaStream (Bilhaut et Widlöcher, 2006).

Unitex[4] (Paumier, 2003), développé au LADL (Laboratoire d'Automatique Documentaire et Linguistique de l'Université Paris 7) puis à l'IGM (Institue Gaspard Monge de l'Université de Marne-la-Vallée) est un système de traitement de corpus qui repose sur des technologies d'automates, distribué sous licence libre LGPL (comme SXPipe). Il permet l'application sur corpus de ressources linguistiques telles que des lexiques et des grammaires (sous forme de réseaux de transitions récursifs). De nombreuses ressources existantes sont utilisables avec Unitex, dont les tables du lexique-grammaire (ou tables du LADL), et ce dans de nombreuses langues, bien que ces ressources ne soient pas toutes librement disponibles. Par rapport à nos besoins, et notablement en vue d'une utilisation comme phase préliminaire à une analyse syntaxique, Unitex souffre de différentes limitations, dont les plus importantes sont :

– pas de correcteur orthographique,

– pas de véritable support de la notion de *cascade* de traitements (rajouter progressivement des informations en utilisant au mieux les informations déjà introduites par les traitements déjà effectués),

– pas de véritable support de l'ambiguïté lors de l'application de grammaires locales (réseaux de transitions récursifs), comme rapporté dans (Danlos, 2005), et comme nous le rappellerons en 7.3,

– efficacité non optimale (bien que la dernière version, la version 2.0b, semble significativement plus rapide que les précédentes).

En revanche, Unitex a un point fort indéniable, qui est son interface d'édition de réseaux de transitions récursifs. C'est une des motivations du travail rapporté en 7.3.

À l'inverse d'Unitex, GATE et LinguaStream sont des plateformes conçues pour appliquer des cascades de traitements, ou modules, tout comme SXPipe. Ils n'ont pas vocation à se limiter aux pré-traitements, et incluent des modules pouvant aller bien au-delà (analyse syntaxique profonde, confrontation avec des ontologies, etc.). GATE (General Architecture for Text Engineering, (Cunningham *et al.*, 2002)), comme son

4. Nous décrivons brièvement Unitex, car c'est un outil que nous avons manipulé. Nous n'exprimons par là aucune préférence particulière entre Unitex et NooJ (`http://www.nooj4nlp.net/`), outil développé à l'Université de Besançon, et qui sont tous deux des successeurs du système INTEX.

nom l'indique, est une plateforme pour le traitement de corpus textuels. C'est tout à la fois une architecture permettant de spécifier des chaînes de traitement à partir de composants, un *framework* écrit en Java pour l'intégration de composants, et une interface graphique de développement de modules et de visualisation des résultats qui repose sur ce *framework*. GATE dispose d'outils standard pour traiter différents formats de corpus, ainsi qu'un certain nombre de modules pré-définis principalement pour le traitement de textes en anglais. En réalité, certains composants sont simplement des *wrappeurs* pour des outils externes. Ainsi, le *plugin* pour le traitement du français inclut uniquement un *wrapper* pour l'étiqueteur `treetagger` et des fonctionnalités élémentaires de traitement de surface[5]. SXPipe n'a pas vocation à concurrencer GATE : c'est plutôt une collection de modules destinés au traitement du français, et dont le but premier est le pré-traitement avant une analyse syntaxique profonde. Toutefois, SXPipe fait explicitement usage de traitement non-déterministes (y compris dans le module de correction orthographique), là où les outils proposés par GATE (et notamment l'architecture JAPE pour la reconnaissance d'expressions régulières) ne sont pas prévus pour cela. De plus SXPipe inclut une architecture originale pour la reconnaissance de motifs non contextuels (et non seulement réguliers), dont nous verrons l'efficacité et l'utilité à la section 7.

La plateforme LinguaStream (Bilhaut et Widlöcher, 2006), développé au GREYC (Université de Caen) partage un certain nombre de principes et de fonctionnalité avec GATE, ainsi que le langage de développement (Java), mais s'en distingue par l'importance accordée à l'utilisation de formalismes déclaratifs et par la possibilité de définir un graphe de traitements (et pas seulement une séquence linéaire). De plus, comme l'indiquent ses développeurs, LinguaStream n'est pas destiné au traitement de corpus très volumineux. Enfin, LinguaStream ne gère pas plus les ambiguïtés que GATE.

L'apport de SXPipe par rapport à ces architectures ne se limite donc pas aux modules qui le composent, bien que certains d'entre eux soient originaux et performants. Avant tout, SXPipe s'en différencie au niveau architectural par sa gestion de l'ambiguïté : un certain nombre de modules de SXPipe produisent des sorties ambiguës, et lisent des entrées ambiguës. Ceci permet à une architecture en cascade (*pipeline*) de modéliser des interactions complexes entre modules. Nous reviendrons plus longuement sur ce point fondamental à la section 4.2. Cette gestion avancée de l'ambiguïté n'est disponible dans aucune des trois architectures citées.

5. Le manuel de GATE décrit ainsi ces fonctionnalités : « *The applications both contain resources for tokenisation, sentence splitting, gazetteer lookup, NE recognition (via JAPE grammars) and orthographic coreference. Note that they are not intended to produce high quality results, they are simply a starting point for a developer working on French* ».

3. Notions de base et formats utilisés

3.1. *Tokens et formes*

Avant de poursuivre, il nous faut préciser quelques termes usuels. Nous utilisons le terme *forme* (ou *forme fléchie*) pour dénoter une unité linguistique syntaxiquement atomique, c'est-à-dire une unité considérée comme élémentaire du point de vue syntaxique (elle est syntaxiquement inanalysable, même si elle peut l'être d'un autre point de vue, en particulier du point de vue morphologique[6]). Il s'agit de la notion de *wordform* au sens du brouillon de norme ISO MAF (Morphosyntactic Annotation Framework, (Clément et de La Clergerie, 2004; Clément et Villemonte de La Clergerie, 2005)). Nous représentons les formes à l'aide de cette police.

Nous utilisons le terme *token* pour dénoter une séquence de caractères présent dans le corpus et séparé de ses voisins par des espaces ou par certaines autres marques typographiques (ponctuation,...) qui varient selon la langue (et qui sont malheureusement parfois ambiguës). Un token est donc ancré dans le texte, il peut être identifié par sa position (ligne, colonne) ou par un identifiant unique. Nous dirons qu'un ou plusieurs tokens *représentent* une ou plusieurs formes : ils les instancient dans le texte. Il s'agit de la notion de *token* au sens de MAF. Nous représentons les tokens à l'aide de `cette police`.

Nous appelons *élément détachable* d'un token la représentation d'une forme juxtaposée à celle d'une autre forme au sein dudit token (exemples : le préfixe `re` dans `restimuler`, les clitiques `-m'` et `en` dans `donne-m'en`). Une fois détachés les éléments détachables, il reste ce que nous appellerons le *noyau* du token.

Enfin, nous appelons *amalgame* un token résultant de la fusion de plusieurs formes (exemple : `duquel`).

Nous appelons *forme simple* une forme dont la représentation ne dépasse pas un token (exemple : article, ou encore la forme représentée par un élément détachable d'un token). Nous appelons *forme composée*, ou, par habitude, *mot composé* voire *composé*, une forme dont la représentation se répartit sur plusieurs tokens (exemples : a contrario). Par convention, une forme composée est représentée en accolant ses composants au moyen du symbole *blanc souligné* (« _ »)[7]. Par convention également,

6. Cela suppose que l'on dispose d'une distinction précise entre morphologie et syntaxe. C'est pourtant loin d'être le cas. En effet, même à l'écrit il est difficile de définir une forme autrement qu'en des termes vagues faisant appel à des notions floues comme l'atomicité syntaxique et/ou sémantique. Ce qui n'aide pas nécessairement à décider combien de formes il y a dans *ceux-ci* (cf. *ceci* ou *ce ... -ci*), *attrape-nigaud*, *aux*, *pomme de terre*, ou *connu comme le loup blanc*. Nous reviendrons sur ce problème à la section 6. Quoi qu'il en soit, la définition de ce qu'est une forme relève en partie d'une convention, et on peut la faire reposer sur le principe pragmatique suivant : est une forme ce qui fait partie du lexique utilisé (par la chaîne de traitement pré-syntaxique et/ou par l'analyseur syntaxique que l'on peut vouloir utiliser en aval de la chaîne).

7. Dans le corpus, les caractères « _ », « { » et « } » sont remplacés par les tokens spéciaux _UNDERSCORE, _ACC_O et _ACC_F, qui sont également des entrées du lexique. Ainsi,

Tokens	DAG de formes	Remarque
`ereurs`	erreurs	correction orthographique
`aujourd'hui`	aujourd'hui	forme simple
`l'idée`	l' idée	deux formes simples
		(`l'` est un élément détachable)
`a priori`	a_priori	forme composée
`pomme de terre`	(pomme de terre	ambiguïté entre formes simples
	| pomme_de_terre)	et forme composée
`duquel`	de lequel	amalgame
`grace au`	(grâce à | grâce_à) le	correction et ambiguïté
`à l'instar du`	à_l'_instar_de le	interaction entre composé et amalgame
`donne-m'en`	donne -m' en	deux éléments détachables
`a-t-elle`	a -t-elle	un seul élément détachable
`12 345,6`	_NUMBER	forme spéciale multi-tokens
`www.inria.fr`	_URL	forme spéciale mono-token

Tableau 1. *Quelques exemples du passage des tokens aux formes.*

les préfixes et les suffixes (qui n'existent pas de façon autonome) sont respectivement suivis et précédés d'un *blanc souligné* (exemples : re-_, sur_, _-né).

On notera que la notion d'amalgame, qui se situe au niveau des tokens, n'est pas incompatible avec celle de composé, qui se situe au niveau des formes. Ainsi, la séquence `à l'instar du` peut être analysée comme représentant la forme à_l'instar_de suivie de la forme le.

Enfin, nous appelons *forme spéciale* une forme qui abstrait une entité nommée que l'on ne veut pas analyser au niveau syntaxique, ou tout autre notion spéciale que l'on souhaite pouvoir exploiter au niveau syntaxique. Par convention, une forme spéciale commence par le symbole *blanc souligné* et se poursuit par des lettres majuscules (exemples : _URL).

Le tableau 1 montre quelques exemples du passage de tokens à formes.

Enfin, nous appelons *lexique orthographique* un lexique contenant les formes simples (qui sont donc des formes), les amalgames et les composants de formes composées. Il s'oppose au *lexique morphologique* qui associe à chaque forme (forme simple ou composée) des informations morphologiques. Ainsi, instar est une entrée du lexique orthographique, mais pas du lexique syntaxique. Ce dernier comprend en revanche une entrée pour à_l'instar_de.

---

ces trois caractères sont disponibles comme méta-caractères, et toute forme qui fait usage du blanc souligné est donc une forme artificielle (forme spéciale, préfixe, suffixe,...).

3.2. *Format utilisé*

Dès leur identification et pour tout le restant du processus, les tokens formant la chaîne d'entrée sont conservés dans des *commentaires* (entre accolades et complétés par leur position dans la chaîne d'entrée) qui précèdent la forme qu'ils représentent. Chaque token est représenté par un élément XML identique aux tokens du format EASy, c'est-à-dire de la forme `<F id="E`i`F`j`">token</F>`, où i est un numéro de phrase et j un numéro de token dans la phrase. Pour alléger les notations, nous représenterons un tel token par $\texttt{token}_j^i$, $\texttt{token}_j$, ou `token` selon les cas, en fonction de l'importance des identifiants dans le contexte.

Pour illustrer ces notations considérons le texte brut suivant.

`contactez-moi au 1 av. Foch, 75016 Paris, ou par e-mail à nom@institut.com.`

Il pourra devenir, en sortie de SXPipe[8] :

{contactez$_1$} contactez {-moi$_2$} moi {au$_3$} à {au$_3$} le {1$_4$ av.$_5$ Foch$_6$,$_7$ 75016$_8$ Paris$_9$} _ADRESSE {,$_{10}$} , {ou$_{11}$} ou {par$_{12}$} par {e-mail$_{13}$} e-mail {à$_{14}$} à {nom@institut.com$_{15}$} _EMAIL {.$_{16}$} .

Cet exemple est néanmoins particulièrement simple, en ce qu'il ne fait apparaître aucune ambiguïté. En effet, la plupart des modules sont susceptibles de produire différentes analyses concurrentes, que l'on peut vouloir conserver en parallèle. C'est le cas par exemple pour certains modules de reconnaissance d'entités nommées ou du module de correction orthographique et d'identification des mots composés. Pour représenter ces ambiguïtés, nous utilisons la notion de DAG (graphe orienté acyclique), sous la forme d'expression régulière (format `dag`) ou de liste de transitions (format `udag`, pour *unfolded DAG* c'est-à-dire *DAG déplié*). Ces deux formats sont illustrés par l'exemple classique que constitue la chaîne `pomme de terre cuite`.

Format `dag` : ({pomme} pomme {de} de {terre cuite} terre_cuite | {pomme de terre} pomme_de_terre {cuite} cuite | {pomme} pomme {de} de {terre} terre {cuite} cuite)

Format `udag` :

```
##DAG BEGIN
 1   {pomme de terre}   pomme_de_terre   4
 1   {pomme}            pomme            2
 2   {de}               de               3
 3   {terre}            terre            4
 3   {terre cuite}      terre_cuite      5
 4   {cuite}            cuite            5
##DAG END
```

Chaque transition est constituée du numéro de l'état de départ de la transition, du

8. On notera que le même token peut être utilisé plusieurs fois de suite, pour gérer les agglutinées (ainsi au$_3$).

commentaire associé, de la forme sur laquelle se fait la transition, et du numéro de l'état final de la transition[9].

Le passage du format `dag` au format `udag` se fait à l'aide d'un outil fourni avec SXPipe, nommé `dag2udag`.

4. Architecture et principes

4.1. *Architecture générale*

Le traitement d'un corpus par SXPipe peut se décomposer en cinq étapes principales[10], comme le montre la figure 1 :

1) traitements au niveau du texte brut ;

2) découpage du texte brut en tokens et en phrases ;

3) traitements au niveau des tokens ;

4) regroupement des tokens en formes (correction orthographique et traitement des composés et des amalgames) ;

5) traitement au niveau du DAG de formes.

Les étapes 1, 3 et 5 sont modulaires : elles comportent divers modules, que l'on peut activer ou non comme expliqué ci-dessous[11]. Les étapes 2 et 4 sont des étapes pivot, en ce sens qu'elles transforment le format du corpus (l'étape 2 transforme du texte en un flux de tokens, l'étape 4 transforme un flux de tokens en un DAG de formes). Tous les modules sont paramétrables, pour s'adapter à différentes applications. Ainsi, à titre d'exemple :

– il est souhaitable de communiquer à chaque module la langue traitée,

9. Il se peut que le `udag` comporte une transition spéciale sur <EOF> arrivant sur l'état final du DAG : la minimalité du DAG fourni (considéré comme un automate fini déterministe) ne permet pas de garantir l'absence de ces transitions sur la chaîne vide, pour peu que l'on souhaite n'avoir qu'un seul état final.

10. Cette séquence de cinq étapes est motivée par le constat suivant : un corpus brut est une séquence de caractères, dans laquelle on veut identifier des formes. Les tokens sont utilisés comme unités intermédiaires pour simplifier certains traitements. On trouvera dans (Heitz, 2006) une discussion sur la problématique de l'ordonnancement des modules dans une chaîne de prétraitement, en particulier pour la recherche de cycles de dépendances entre traitements. Toutefois, les dépendances entre traitement décrites par l'auteur sont parfois surprenantes, tout comme l'ordre d'exécution des traitement qu'il en induit. Ainsi, la correction orthographique et la segmentation « finale » en mots sont respectivement effectués au tout début et à la fin du processus, alors qu'ils sont inextricablement liés. Par ailleurs, on voit mal où pourraient s'insérer, dans l'architecture proposée, des modules aussi simples et indispensables que la reconnaissance d'URL, que l'auteur place dans une famille à exécuter *après* la correction orthographique.

11. Il existe dans certains cas des dépendances entre modules d'une même étape, en ce sens que certains modules doivent être utilisés avant d'autres.

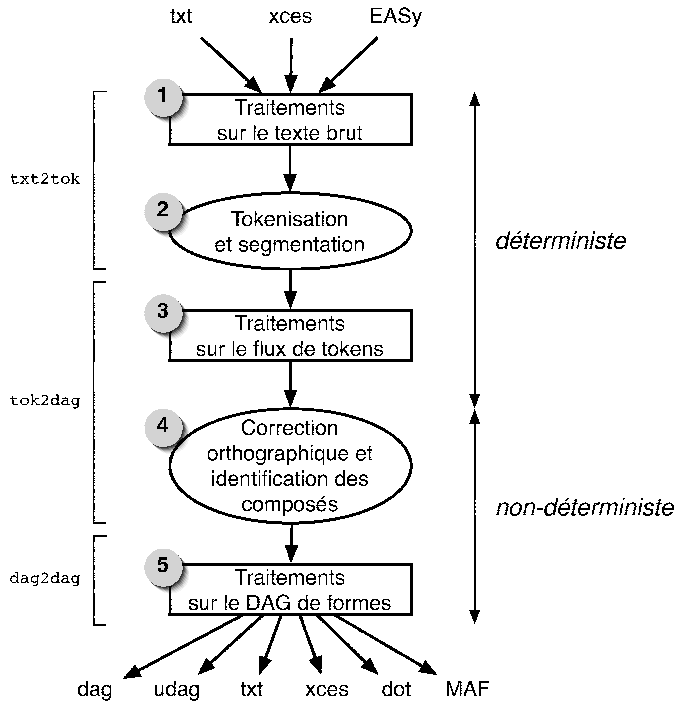


Figure 1. *Architecture globale du système.*

– l'activation de certains modules peut être utile ou non selon les cas : les grammaire de reconnaissance des URL et des adresses courriers électroniques n'est pas indispensable au traitement d'un corpus littéraire du XIX$^{\text{ème}}$ siècle ;

– le paramétrage de certains modules peut varier selon les cas : le traitement d'un corpus journalistique de qualité devra se faire avec un correcteur orthographique strict (tout ce qui ne peut se corriger à faible coût doit être considéré comme un mot inconnu), alors que le traitement d'un corpus de courriers électroniques nécessitera un paramétrage plus souple (pour corriger des mots très mal orthographiés).

Chaque utilisateur peut donc se construire sa propre chaîne de traitement, avec ses propres paramètres et jeux de modules. Pour cela, il suffit de définir un fichier de configuration, qui liste pour chaque étape les modules à utiliser (un seul module, obligatoire, correspond aux étapes pivot 2 et 4), ainsi que les paramètres à leur fournir. L'outil `lingpipe`, fourni dans SXPipe, est alors en mesure d'exécuter la chaîne ainsi définie, ou d'en faire un *script* exécutable.

Naturellement, chaque utilisateur peut également construire ses propres modules, qui doivent relever de l'une des étapes 1, 3 et 5. Ils peuvent alors être intégrés à l'architecture, et partagés avec les autres utilisateurs.

Dans la distribution informatique de SXPipe 2, un certain nombre de modules et de chaînes pré-définis sont proposées. Outre la chaîne `sxpipe`, chaîne par défaut pour le traitement du français, sont incluses les chaînes `sxpipe-en`, `sxpipe-es`, `sxpipe-pl`, `sxpipe-sk` et `sxpipe-si`, chaînes par défaut pour le traitement respectif de l'anglais, de l'espagnol, du polonais, du slovaque et du slovène. On notera toutefois que la compilation de SXPipe ne produit un *script* exécutable donné qu'à la condition que le lexique Alexina[12] de la langue qu'il traite soit déjà installé.

En préliminaire aux traitements effectués lors des étapes 1 à 5, on peut avoir à utiliser un module d'extraction du texte brut à partir d'un autre format (format EASy, format XCES)[13] et/ou un module de détection automatique de la langue. Par ailleurs, en aval de la chaîne, des traitements complémentaires peuvent également être appliqués, par exemple pour transformer le DAG produit en texte, en corpus au format XCES, voire en représentation graphique.

Après une réflexion sur la prise en compte des ambiguïtés dans SXPipe, les prochaines sections de cet article passeront en revue ces traitements, en suivant pour cela l'architecture en sous-dossiers structurant le paquet informatique téléchargeable sur son site internet. Ainsi, les étapes 1 et 2, regroupées dans le dossier `txt2tok`, font l'objet de la section 5. Les étapes 3 et 4, regroupées dans le dossier `tok2dag` sont décrites à la section 6. Enfin, l'étape 5, qui correspond au dossier `dag2dag`, est étudiée à la section 7.

4.2. *Chaîne de traitement et ambiguïtés*

Dans l'idéal, une chaîne de traitement comme SXPipe doit allier deux caractéristiques en apparence contradictoires :

modularité pour permettre une souplesse d'utilisation et simplifier le développement de la chaîne,

prudence pour éviter que des décisions prises trop tôt ne conduisent ultérieurement à des erreurs irréversibles (cf. exemples ci-dessous).

Pour être à la fois prudente et modulaire, une chaîne doit être composée de plusieurs modules dont chacun sait reporter à un module ultérieur des décisions pour lesquelles il ne dispose pas de toutes les informations nécessaires. Il faut donc idéa-

12. Alexina est une architecture linguistique et informatique pour le développement de lexiques morphologiques et syntaxiques. C'est l'architecture sur laquelle repose le Le*fff*, lexique à large couverture du français sur lequel nous reviendrons. Le site internet d'Alexina est `http://alexina.gforge.inria.fr/`.

13. Dans ces deux cas, le découpage en tokens est déjà effectué, et doit être conservé tel quel. Ceci constitue une contrainte pour SXPipe, qui doit être ainsi adapté pour appliquer ses traitement malgré une tokenisation qui n'est pas celle qu'il aurait produit lui-même. Dans la distribution informatique de SXPipe, la préservation d'une tokenisation fournie en entrée se fait, au niveau des modules, par l'option `-sw`, ou plus globalement par l'appel de la chaîne pré-définie `sxpipe-easy` (et non `sxpipe`, chaîne par défaut traitant le français).

lement que chacun des modules sache prendre en entrée et produire en sortie des informations *ambiguës*. C'est ce que SXPipe propose, non pour tous ses modules, mais pour ceux qui sont le plus susceptibles de faire face à des phénomènes ambigus. Ce n'est pas le cas de tous les systèmes comparables, qui ont donc des difficultés à traiter correctement, et dans le cas général, des exemples d'ambiguïté pourtant bien connus (cf. section 2).

Illustrons ceci sur quelques exemples classiques. Supposons qu'un module décide par exemple que `trente et un`, situé au milieu d'une phrase complète, est toujours un nombre. Cette décision irrévocable, parfois erronée, pourra conduire à des problèmes au cours de traitement ultérieurs effectués par SXPipe ou par des outils utilisés en aval (comme des analyseurs syntaxiques). C'est le cas dans une phrase comme `...au cours des années trente et un peu après`.

En réalité, les ambiguïtés peuvent concerner la majorité des modules, comme le montrent les quelques exemples ci-dessous :

correction orthographique : `pese` est pesé dans `je l'ai pese`, mais pèse dans `ca ne pese pas lourd`[14] ;

traitement des composés : `grâce à` est grâce_à dans `c'est grâce à toi` mais grâce à dans `ils rendent grâce à leur créateur` ; `du` est du dans `manger du pain` mais de le dans `parler du pain` ;

entités nommées : outre l'exemple ci-dessus sur les nombres, on peut proposer l'exemple suivant : `le 18 courant` est une expression temporelle dans suite à votre lettre {du} de {du} le 18 courant[15], mais ne l'est pas dans il a été vu le 18 courant un marathon (dans ce cas, seul `le 18` est une expression temporelle).

Il existe trois façons d'aborder ce problème :

1) en faire fi, au prix d'un traitement incorrect de cas tels que ceux que nous venons de montrer ;

2) rassembler tous les modules en un seul module très complexe, au développement délicat, mais qui est à même de traiter toutes les interactions entre tous les modules ;

3) permettre à chaque module de prendre en entrée et de produire en sortie non pas une séquence linéaire de tokens ou de formes, mais un DAG de tokens ou de formes.

SXPipe fait successivement usage de ces trois façons de faire :

– les modules de SXPipe traitant du texte brut et des tokens sont déterministes (solution 1), car les grammaires concernées semblent ne pas devoir produire d'ambiguïté,

14. Nous ne prétendons pas signifier par cet exemple qu'il est impossible de déterminer automatiquement la bonne correction, par exemple en se servant du contexte gauche. Cet exemple est ici simplement pour illustrer le fait que la correction orthographique *peut* être source d'ambiguïtés.

15. Seul le commentaire associé aux formes de et le a été indiqué, car c'est le seul cas, dans cet exemple, où tokens et formes (données en entrée du module) diffèrent.

– le module identifiant les formes à partir de la séquence de tokens, module qui effectue à la fois la correction orthographique et le traitement des composés, amalgames et autres formes détachables peut être vu comme relevant de la solution 2,

– tous les traitements effectués sur le DAG de formes ainsi produit, et qui modifient ou enrichissent ce DAG, gèrent parfaitement l'ambiguïté, en entrée comme en sortie : ils appliquent la solution 3.

Les trois prochaines sections décrivent plus en détail les composants de SXPipe. La section 5 décrit les grammaires *perl* agissant sur le texte brut ainsi que le segmenteur en tokens et en phrases. S'il ne s'agit pas là de modules très innovants, ils n'en sont pas moins indispensables. La section 6 décrit les grammaires agissant sur les tokens, mais surtout le correcteur orthographique et identifieur de formes composés, qui repose sur une implémentation efficace des automates finis, des règles de correction et des heuristiques complexes. La section 7 décrit l'architecture originale de reconnaissance de motifs non contextuels à l'aide du système SYNTAX. Des évaluations de certains de ces modules sont proposées au fur et à mesure de leur description.

5. Du texte aux tokens : motifs dans le texte brut, tokenisation, segmentation

5.1. *Entités nommées au niveau du caractère*

Les corpus réels ne sont pas comme les phrases de linguistes. Ils comportent des séquences de caractères ou de tokens qui ne sont analysables ni morphologiquement ni syntaxiquement, mais procèdent de motifs productifs. Ces séquences sont généralement appelées *entités nommées*. Nous utilisons ce terme dans un sens légèrement plus large que le sens habituel (Maynard *et al.*, 2001), en y incluant non seulement les dates, noms propres et autres « vraies entités » nommées, mais également les nombres, les séquences en langues étrangères et d'autres. Nous appelons *grammaire locale* une grammaire qui reconnaît une certaine famille d'entités nommées.

Parmi ces entités, certaines contiennent des caractères qui sont habituellement des marques de ponctuation, en particulier le point (par exemple dans les URL), mais aussi la virgule (dans les adresses) ou toute sorte d'autres caractères servant généralement de ponctuation (par exemple dans les *smileys*). Il en va ainsi d'entités nommées comme les adresses (postales ou électroniques), les URL, les nombres écrits à l'aide de chiffres, et d'autres encore. Les grammaires locales reconnaissant ces entités nommées doivent donc être appliquées avant même la tokenisation et la segmentation en phrases, directement sur le texte brut. Ces grammaires locales, qui agissent donc au niveau du caractère, font partie de l'ensemble de modules `txt2tok`.

Nous avons ainsi développé un ensemble de grammaires locales robustes[16] agissant au niveau du caractère, implémentées sous la forme de programmes *perl* mettant

16. Par robuste, nous voulons dire que les entités nommées comportant des erreurs sont également reconnues, comme par exemple `ttp :/url.avec.erreurs.com /index.html`.

en jeu un grand nombre d'expressions régulières. Ces grammaires associe aux entités reconnues une forme spéciale.

Considérons par exemple l'exemple suivant.

```
Plus d'informations sur http://www.siteweb.fr.
```

L'application de la grammaire locale des URL produira le résultat ci-dessous[17].

```
Plus d'informations sur {http://www.siteweb.fr} _URL .
```

À ce stade, le format de représentation des tokens n'est pas définitif: l'enrobage dans des éléments XML y est absent, et, au sein d'un commentaire, le caractère espace dénote une frontière entre tokens. C'est le module de tokenisation et segmentation en phrase, décrit ci-dessous, qui parachèvera la construction des tokens.

5.1.1. *Grammaires locales*

Dans la version actuelle de SXPipe 2, les grammaires locales agissant au niveau du caractère reconnaissent les phénomènes suivants:

adresses e-mail avec détection des espaces erronés (forme spéciale _EMAIL),

URL avec détection de nombreux cas d'erreurs et de nombreux formats[18] (_URL),

dates sous différents formats ainsi que les intervalles de dates: par exemple, `du 29 au 31 janvier` devient `du` _DATE_arto `au` _DATE_arto, alors que `29`, lorsqu'il est isolé, n'est reconnu que comme un nombre (formes spéciales _DATE_arto, _DATE_artf, _DATE_year qui permettent de différencier les comportements syntaxiques d'expressions telles que respectivement `29 janvier`, `lundi prochain` et `1978`),

numéros de téléphone dans divers formats (_TEL),

horaires dans de nombreux formats ainsi que les intervalles horaires (par exemple, `2-3 heures`, `3 ou 4 minutes`, etc. — forme spéciale _HEURE),

17. On constate que le point final a été correctement identifié comme ne faisant pas partie de l'URL. Toutefois, l'information selon laquelle il était accolé, sans séparateur, au dernier caractère représentant cette URL a été perdue dans le résultat, au moins sous sa forme montrée ici. En réalité, l'option `-sw`, déjà évoquée, permet de préserver cette information. Dans ce cas, le point est remplacé par `_REGLUE_.`, où `_REGLUE_` indiquera à un traitement ultérieur de normalisation qu'il faut ré-assembler `http://plus.dinformations.fr`et le point en un seul token, qui servira de commentaire à la fois à `_URL` et à la forme « point ». Il existe de même un autre mécanisme, `_UNSPLIT_`, permettant de partager un commentaire (un ou plusieurs tokens) entre deux formes. Pour schématiser et illustrer:
`{c}` a `_REGLUE_b` deviendra `{<F id="EiFj">cb</F>}` a `{<F id="EiFj">cb</F>}` b
`a _UNSPLIT_b` deviendra `{<F id="EiFj">a</F>}` a `{<F id="EiFj">a</F>}` b.

18. Une attention particulière a été apportée au problème suivant: on rencontre en corpus des cas où l'espace est remplacé par un point, pour séparer deux mots. Dans ce cas, si le deuxième des mots concernés ne fait que deux lettres, il peut être confondu avec une extension valide de nom de domaine (ainsi, `de` est l'extension des sites internet allemands). Il faut donc être plus prudent, quand on traite du français, dans la reconnaissance des site internet allemands (entre autres cas de ce type).

adresses postales dans de nombreux formats différents (_ADRESSE),

phénomènes particuliers :

– nombres faisant partie de formes composées (`France 2` devient France_2)[19]

– répétitions et hésitations, par exemple pour essayer de normaliser les transcriptions de corpus oraux (`le euh le la le verre de de vin` devient {`le euh le la le`} le `verre` {`de de`} de `vin`)[20],

– mots transcrits sous la forme `incomp(let)`, qui deviennent {`incomp(let)`} incomplet,

– quelques abréviations courantes (`bcp`, `ns`, `nbreux`, mots en `-t^...`),

– quelques typos courantes et délicates à corriger de façon générique (`à priori` pour a priori, `er` pour et...),

– mots entre guillemets (`un «test»` devient `un` {`«test»`} test[21]),

– phrases commençant par une séquence en majuscules, alors que le reste ne l'est pas: la séquence en majuscule est identifiée comme étant un titre, à séparer de la suite (fréquent dans les corpus journalistiques),

préfixes numériques servant à la construction de mots savants: `3-folié` devient {`3-`} _NPREF `_-folié`,

nombres en chiffres sous différents formats, ainsi que les ordinaux comportant des chiffres, tels que `2ème` (forme spéciale _NUMBER), et les en-têtes d'éléments de listes (de type `2-` ou `*`), formes spéciales _META_TEXTUAL_GN et _META_TEXTUAL_PONCT),

smileys tels que `:-)` ou `:D` (_SMILEY),

artefacts de formatage pour traiter divers phénomènes spéciaux de formatage textuel, particulièrement utilisé dans les corpus de courrier électronique (séquences du type `*très important*` ou `_vraiment capital_`).

Naturellement, une partie non négligeable de ces grammaires locales dépend de la langue, en particulier les listes de « mots » qui prennent part aux motifs. Ainsi, une fois définis pour chaque langue des motifs reconnaissant les mois, les jours de la semaine, et d'autres encore, certains motifs reconnaissant des dates sont indépendants

19. Dans `en France, 2 personnes sur 3...`, la virgule évite une possible ambiguïté. Il n'est pas exclu que certains cas soient véritablement ambigus, alors que les grammaires locales de `txt2tok` sont déterministes. On est ramené à la discussion du paragraphe 4.2.

20. Rappelons que l'on peut choisir d'utiliser une chaîne personnalisée, par exemple une chaîne qui ne fait pas appel à ce module. En effet, l'utilisation d'un analyseur syntaxique développé pour des corpus textuels en vue de l'analyse de transcription de corpus oraux nécessite de tels traitements au préalable. Ce n'est pas nécessairement le cas, au contraire, si on fait usage d'un analyseur syntaxique développé spécifiquement pour les corpus oraux.

21. Dans un cas comme celui-ci, test n'est pas encore à proprement parler une forme: il passera par le correcteur orthographique et reconnaisseur de composés, seul habilité à produire un DAG de formes.

Corpus	general_elda		oral_elda_1		mail_1	
	#motifs	précision	#motifs	précision	#motifs	précision
courrier électr.	6	100%	0	–	0	–
URL	27	100%	1	100%	0	–
date	38	100%	37	100%	43	98%[22]
horaire	0	–	12	100%	40	100%
adresse	14	100%	0	–	0	–
smiley	0	–	0	–	84	100%
ph. particuliers	21	100%	197	100%	375	100%
nombres	275	100%	168	100%	101	100%
formattage	3	100%	0	–	0	–
Nombre de tokens[23]	*16 786*		*11 987*		*13 717*	
Temps d'exécution	*3,61s*		*3,06s*		*2,77s*	

Tableau 2. *Évaluation de différents modules de* `txt2tok`.

de la langue. Il en reste qui sont spécifiques à une langue donnée. Il en va de même pour la plupart des autres grammaires locales.

5.1.2. *Évaluation des grammaires locales*

Pour évaluer manuellement ces grammaires locales, nous avons utilisés différents sous-corpus du corpus EASy, utilisé dans le cadre de la campagne EASy d'évaluation des analyseurs syntaxiques. L'intérêt du corpus EASy est qu'il comporte des sous-corpus de genres très variés (journalistique, littéraire, technique, courriers électroniques, transcription de corpus oraux, corpus de questions, etc.).

Nous avons choisi trois types de sous-corpus, aussi variés que possibles: un sous-corpus de pages internet (sous-corpus `general_elda`), un corpus de transcription de corpus oraux (sous-corpus `oral_elda_1`) et un corpus de courriers électroniques (sous-corpus `mail_1`). Ce sont en effet parmi ces types de corpus que l'on est susceptible de trouver le plus grand nombre d'entités nommées telles que reconnues par les modules présentés ci-dessus.

Les résultats sont présentés dans le tableau 2.

5.2. *Tokenisation et segmentation en phrases*

Après avoir appliqué ces grammaires locales, la phase `txt2tok` se termine par la tokenisation et la segmentation en phrases. Cette tâche est réalisée par un ensemble

23. L'unique erreur est la suivante: `27 , 28 , 29 juin` est devenu `27 , 28 , {29 juin} _DATE`. Naturellement, ayant constaté cette erreur, la grammaire a été améliorée.
23. Il s'agit des tokens fournis par le format EASy.

volumineux d'expressions régulières *perl* qui étendent les idées simples proposées par exemple par (Grefenstette et Tapanainen, 1994), et qui prennent en compte la liste des mots connus du lexique comportant un point (souvent des abréviations). SXPipe traite correctement toutes sortes de faux positifs et de faux négatifs qui apparaissent inévitablement dans un corpus réel. À l'issue de cette étape, un retour à la ligne matérialise une frontière de phrase certaine, et la forme spéciale _SENT_BOUND matérialise une frontière de phrase incertaine (ou absente d'une segmentation en phrase déjà fournie en entrée, comme dans le cas du traitement d'un corpus au format EASy).

6. Des tokens aux formes: correction orthographique et reconnaissance des mots composés

6.1. *Entités nommées au niveau des tokens*

Une fois les tokens identifiés, nous allons appliquer un certain nombre de grammaires agissant au niveau des tokens. Pour cela, nous commençons par décorer le texte à l'aide d'une information complémentaire: le correcteur orthographique décrit dans la section suivante est utilisé dans un mode dégradé, en ce sens qu'aucune correction orthographique n'est faite, mais que l'on marque les *tokens inanalysables*, c'est-à-dire les tokens qui ne peuvent pas être analysés comme des formes connues (présentes dans le lexique) ou des combinaisons de formes connues (comme anticommuniste-né, donne-m'en ou scénario-catastrophe).

Une fois identifiés les tokens inanalysables (au sens du paragraphe précédent), interviennent alors des grammaires locales agissant au niveau des tokens et tirant parti de cette information supplémentaire. Elles reconnaissent les classes suivantes d'entités nommées:

acronymes suivis ou précédés de leur expansion, avec diverses variantes typographiques possibles (forme spéciale _NP_WITH_INITIALS),

noms propres précédés par un titre (comme `Dr.` ou `Miss`, forme spéciale _NP),

séquences en langue étrangère autre que la langue traitée (forme spéciale _ETR).

Les deux dernières grammaires locales méritent plus d'explication. Elles reposent sur la technique suivante. Soit $t_1 \dots t_n$ une phrase dont les tokens sont les t_i. Nous définissons une fonction d'étiquetage e qui associe (grâce à des expressions régulières) une étiquette $e_i = e(t_i)$ à chaque token t_i, où les e_i sont pris dans un petit ensemble fini d'étiquettes possibles (respectivement 9 et 12 pour les deux grammaires locales concernées). Ainsi, une séquence d'étiquettes $e_1 \dots e_n$ est associée à $t_1 \dots t_n$. Ensuite, un (gros) ensemble d'expressions régulières transforme $e_1 \dots e_n$ en une nouvelle séquence d'étiquettes $e'_1 \dots e'_n$. Si dans cette dernière la sous-séquence $e'_i \dots e'_j$ correspond à un certain motif de référence, la séquence de tokens correspondante $t_i \dots t_j$ est considérée comme reconnue par la grammaire locale.

Considérons par exemple l'énoncé

```
Peu après , le Center for irish Studies publiait ...,
```

où `Center`, `irish` et `Studies` ont été identifiés comme tokens inanalysables. On associe à cet énoncé la série d'étiquettes `cnpNEEucn...`[24]. Des expressions régulières sur ces étiquettes mènent à `cnpNeeeen...`, où `e` correspond à *étranger*, ce qui signifie que `Center for irish Studies` est reconnu comme une séquence en langue étrangère[25]. La phrase devient alors (_ETR correspond à *séquence en langue étrangère*):

`Peu après , le {Center for irish Studies} _ETR publiait ...`

6.2. *Correction orthographique ambiguë et reconnaissance des mots composés*

La prochaine étape dans notre chaîne de traitement est le correcteur orthographique et reconnaisseur de formes composées.

Les corpus réels ont des taux variables de fautes d'orthographe, qui vont de quasiment zéro (p. ex. dans les corpus littéraires) à des taux très élevés (p. ex. dans les corpus de courrier électronique). De plus, s'ils restent non corrigés, les mots mal orthographiés deviennent des mots inconnus pour les traitements utilisés en aval de SXPipe: analyseur syntaxique, module d'extraction d'informations, etc. Ceci doit être évité au maximum. À titre d'exemple, dans le cas d'un analyseur syntaxique, les mots inconnus se voient attribuer des informations syntaxiques par défaut qui conduisent à la fois à une faible précision et à une très forte ambiguïté au niveau lexical, qui se répercutent au niveau syntaxique. Dans un contexte d'extraction ou de recherche d'informations, des mots mal orthographiés diminuent le rappel et la qualité de l'indexation des documents[26].

Cependant, la non-correspondance entre tokens et formes rend la correction orthographique délicate. Il faut en effet savoir distinguer un token qui est le résultat de l'accumulation de formes correctes (par amalgame, préfixation, suffixation et/ou apposition) d'un token qui est le résultat d'une ou de plusieurs erreurs (de frappe,

---

24. `c` correspond à *initiale en majuscule*, `n` à *probablement français* (cas par défaut), `p` à *ponctuation*, `N` à *connu comme français*, `E` à *connu comme étranger* et `u` à *token inanalysable*.

25. Nous avons également développé un outil prototype d'identification de la langue pour de telles séquences. Dans cet exemple, la réponse correcte — anglais — est trouvée.

26. Certains analyseurs syntaxiques disposent d'un module de traitement des mots inconnus qui reposent sur la notion d'identification floue (*fuzzy matching*). Cependant, de telles techniques ne permettent pas, dans le cas général, de traiter correctement tous les cas possibles (mots collés), à moins de constituer un module de correction orthographique à part entière. Mais disposer d'un module de correction orthographique dans le pré-traitement permet de s'en dispenser dans l'analyseur syntaxique et d'en modulariser le développement. Il permet également aux tâches ultérieures du pré-traitement de bénéficier des corrections effectuées (cf. section 7). De plus, certains traitements appliqués après SXPipe (reconnaissance d'entités nommées complexes...) utilisent souvent des outils qui ne disposent pas de module de gestion des mots inconnus. Enfin, certains traitements (normalisation de textes...) nécessitent une phase de correction orthographique mais n'utilisent aucun outil en aval de SXPipe, qui serait susceptible de gérer convenablement les mots inconnus.

d'orthographe, d'OCR ou autre). La situation se complique lorsque l'on veut pouvoir gérer correctement les tokens qui sont à la fois le résultat de l'accumulation de plusieurs formes et d'une ou de plusieurs erreurs. Tout ceci est encore plus délicat lorsque l'on souhaite reconnaître les possibles formes composées, voire les espaces (frontières de tokens) insérés par erreur dans le texte brut.

L'expérience montre que la seule façon satisfaisante de traiter ces problèmes est de les traiter simultanément, en préservant les ambiguïtés dès lors que l'on ne dispose pas des informations permettant de les lever. C'est le rôle du module TEXT2DAG, décrit dans cette section. Conformément aux principes généraux qui sous-tendent SXPipe (cf. section 3), TEXT2DAG conserve le ou les tokens de départ (eventuellement mal orthographié) dans les commentaires, et produit une ou plusieurs formes corrigées. Ainsi, TEXT2DAG transformera une séquence de tokens telle que `séquance avecerreurs` sera convertie en {`séquance`$_{1..2}$} séquence {`avecerreurs`$_{2..3}$} avec {`avecerreurs`$_{2..3}$} erreurs.

TEXT2DAG reposant sur un gestionnaire de corrections orthographiques, nommé SXSpell, nous commençons par décrire ce dernier. Puis nous détaillons le fonctionnement de TEXT2DAG lui-même. L'ensemble repose sur le Le*fff*, lexique morphologique et syntaxique du français à large couverture (Sagot *et al.*, 2006)[27].

On notera que TEXT2DAG est très immédiatement adaptable à une autre langue, pour peu que l'on dispose d'un lexique orthographique pour la langue donnée. Bien sûr, comme nous le verrons, les règles de correction peuvent dépendre de la langue, mais des règles génériques donnent des résultats déjà satisfaisants. Et l'ajout de règles particulières est très simple. À l'inverse, aucune correction grammaticale (accord, verbe support, etc.) n'est effectuée.

6.2.1. *Correction orthographique de base:* SX*Spell*

6.2.1.1. Mécanismes sous-jacents

De nombreux travaux ont déjà été réalisés dans le domaine de la correction orthographique (cf. par exemple (Oflazer, 1996) ou la synthèse de (Kukich, 1992)). Les techniques de correction des mots pris isolément se regroupent principalement en deux catégories: les techniques avec entraînement et les techniques sans entraînement. Les principales techniques avec entraînement sont les techniques stochastiques (souvent fondées sur les n-grammes) et les réseaux de neurones. Les techniques sans entraînement reposent sur la *distance minimum de correction* (ci-après MED, pour *minimum edit distance*), qui met en œuvre les opérations élémentaires d'insertion, de suppression, de remplacement et d'interversion, ou sur des *règles* (fondées sur des règles de réécriture qui peuvent être dépendantes du contexte et dont l'origine vient de la phonologie à états finis). Les règles sont clairement plus puissantes et mieux adaptées que la MED[28], mais les opérations citées peuvent également être utiles en tant que telles.

27. Ici, les informations syntaxiques sont naturellement superflues.
28. Un exemple simple en est le fait suivant. Le son [o] pouvant s'écrire (entre autres) *o* ou *eau*, il est raisonnable de savoir corriger l'un en l'autre. Il est alors plus naturel et plus approprié par

Ainsi, notre correcteur est un correcteur à règles, mais les opérations MED habituelles sont également disponibles pour écrire des règles sous-spécifiées.

Le résultat de l'application d'une règle est appelée *modification élémentaire* du mot de départ. Dans le cas général, une *modification* du mot de départ peut être le résultat de plusieurs modifications élémentaires. Toutefois, ces dernières doivent s'être appliquées en des endroits différents du mot de départ, puisque ce n'est que sur ce mot de départ que l'on effectue des modifications élémentaires[29]. Pour un mot donné, chacune de ses modifications est recherchée dans un (ou plusieurs) lexique(s) orthographique(s). Si elle y figure, cette modification est appelée *correction* du mot de départ.

Nous associons à chaque règle un *poids local* et un *poids de composition*. Le coût total d'une modification est calculé de la façon suivante: chaque nouvelle modification élémentaire le coût total de son poids local; par ailleurs, si cette modification élémentaire n'est pas la première effectuée sur le mot en cours de traitement, et sauf indication contraire associée à la correction appliquée, le coût total est augmenté du poids de composition. Ceci permet d'avoir un coût total plus élevé que la somme des coûts locaux. Une fois identifiées celles des modifications qui sont véritablement des corrections (elles sont dans le lexiques), ces corrections peuvent donc être classées par coût croissant, les meilleures corrections ayant les coûts les plus faibles[30]. Naturellement, si un mot est dans le lexique il a une correction de coût nul, à savoir lui-même. On notera que le coût le plus bas parmi les corrections peut correspondre à plus d'une correction.

Notre objectif était de disposer d'une implémentation efficace de ces techniques simples, même en utilisant des règles réalistes et donc nombreuses et un lexique orthographique couvrant construit à partir du Le*fff* (notre lexique orthographique pour le français a plus de 400 000 entrées). Pour atteindre cet objectif, nous considérons:

– le lexique orthographique comme un automate fini déterministe $\mathcal{F}$,

– le mot d'entrée w comme un automate fini linéaire pondéré $\mathcal{T}_w^0$ dont toutes les transitions sont de poids nul,

– les règles de réécriture comme des transducteurs finis pondérés $\mathcal{R}^i (i > 0)$.

L'application de $\mathcal{R}^i$ à tous les endroits possibles de $\mathcal{T}_w^0$ produit un automate fini pondéré $\mathcal{T}_w^i$ rassemblant toutes les modifications produites par la règle i à partir du mot de départ (si donc i ne peut s'appliquer nulle part, $\mathcal{T}_w^i = \mathcal{T}_w^0$). On calcule alors l'automate fini pondéré $\mathcal{T}_w^{all}$, union de tous les $\mathcal{T}_w^i$. Il représente donc l'ensemble de toutes

rapport aux calculs de coût de correction de voir cette opération comme un remplacement de *eau* par *o* que de la considérer comme la succession de deux suppressions et d'un remplacement.

29. Autrement dit, nous n'autorisons pas une règle à modifier le résultat d'une règle précédemment appliquée. Ceci garantit la convergence du processus et l'indépendance du résultat à l'ordre d'application des règles.

30. Dans un avenir proche, nous comptons intégrer à la pondération des corrections proposées des informations fréquentielles (fréquence lexicale ou modèles n-grammes). Les résultats devraient en être améliorés.

les séquences possibles de caractères qui peuvent être obtenues à partir de w par l'application des règles, ainsi que leur coût. Bien sûr, un seuil peut être donné en paramètre pour empêcher le calcul de corrections déraisonnablement coûteuses. Puis nous extrayons de $\mathcal{T}_w^{all}$ tous les mots qui existent effectivement dans le lexique, en calculant l'intersection[31] de $\mathcal{F}$ avec $\mathcal{T}_w^{all}$. Selon ce que demande l'utilisateur (ou le programme faisant usage de SXSpell), on fournit alors la meilleure correction, les corrections de coût optimal voire les n premières corrections dont le coût fait partie des m meilleurs coûts.

La difficulté de cette approche n'est pas la théorie sous-jacente, qui est bien connue, mais vient de la taille des automates à manipuler. En effet, avec un nombre de règles de l'ordre de quelques centaines, l'automate $\mathcal{T}_w^{all}$ a facilement des milliards de chemins. Et il doit être intersecté avec $\mathcal{F}$ et ses 400 000 chemins. Pour cette raison, nous avons utilisé de manière intensive des techniques de tabulation et de représentation compacte, qui reposent en partie sur la bibliothèque de structures de données et de fonctions qui fait partie du logiciel SYNTAX[32]. Les résultats fournis plus bas montrent l'efficacité du correcteur obtenu.

6.2.1.2. Règles de correction

SXSpell n'est toutefois rien sans un jeu approprié de règles de corrections. Nous avons développé différents jeux de règles (pondérées manuellement), que l'on peut activer ou désactiver en fonction des besoins. On peut regrouper ces règles en quatre familles:

– règles « universelles » (exemple: passage minuscule-majuscule ou inversement, interversion de deux lettres,...);

– règles spécifiques à un contexte particulier (exemples: `1` peut se transformer à faible coût en `l` ou `i` dans un contexte d'OCR; `d` peut se transformer à coût modéré en un caractère de la liste `erfcxs` dans un contexte de correction de fautes de frappe, car ce sont les touches voisines sur un clavier informatique[33]);

– règles dépendant de l'encodage (exemple: ajout d'un diacritique sur une lettre; un texte en encodage *latin-1* pourra se voir appliquer une règle remplaçant `e` par une des lettres de la liste `èéêë`, alors que pour un texte en *latin-2* on aura comme caractères cibles la liste `ęěë`);

– règles spécifiques à une langue donnée (exemples: échange de `o`, `au` et `eau` en français, échange de `rz` et `ż` en polonais, échange de `u` en `v` en slovène,...)

---

31. La réalité est plus complexe, en particulier parce que la correction d'une chaîne comportant des espaces doit d'abord essayer d'y identifier plusieurs mots et non un seul.
32. SYNTAX est un système de génération d'analyseurs syntaxiques très efficaces développé en grande partie par Pierre Boullier (Boullier et Deschamp, 1988). Librement disponible, SYNTAX est utilisé pour la reconnaissance de motifs non contextuels. Pour cette raison, nous reportons à la section 7 une description plus complète.
33. On notera que le téléphone iPhone d'Apple met en œuvre cette idée.

Les règles gérées par SXSpell ne sont pas nécessairement des règles remplaçant un caractère par un autre, mais peuvent également remplacer toute chaîne de caractères par une autre. De plus, les caractères spéciaux ^ et $ permettent d'indiquer le début ou la fin du mot, comme il est d'usage dans la représentation des expressions régulières.

Enfin, différents jeux de règles permettent de définir différents niveaux de correction. Dans la suite, nous utiliserons ainsi ce que nous appelons une *correction légère*, qui ne met en jeu que des règles simples, et une *correction complète*, qui met en jeu toutes les règles jugées pertinentes.

6.2.2. *Correction orthographique globale et reconnaissance des formes composées:* TEXT2DAG

Toutefois, comme indiqué précédemment, cette correction orthographique de base ne convient pas telle quelle à la correction d'un token. En effet, au moins trois phénomènes sont à prendre en compte:

– les formes détachables qui sont la conséquence de morphologie dérivationnelle productive (par exemple `anti-Bush`) ou de l'agglutination syntaxique (par exemple `préchoisis-t'en` qui doit être analysé en pré-_ choisis t' en),

– les fautes d'orthographe (ou typographiques) concernant plusieurs tokens (ainsi `corre ction` au lieu de `correction`)

– les fautes d'orthographe (ou typographiques) concernant plus d'une forme (par exemple `unproblème` au lieu de `un problème`).

Enfin, un token commençant par une majuscule, composé exclusivement de majuscules, et/ou situé en début de phrase devra nécessiter des traitements particuliers.

TEXT2DAG sait prendre en compte ces phénomènes et faire appel au correcteur de mots isolés SXSpell, afin de découper, regrouper et/ou corriger les tokens en formes, tout ceci pouvant se faire de façon ambiguë. Des mots inconnus peuvent rester si aucune correction n'est trouvée dont le coût soit inférieur à un certain seuil. Ils sont alors laissés tels quels, pour que les phases ultérieures de traitement (cf. section suivante) puissent s'appuyer dessus, avant, le cas échéant, de les remplacer par des formes spéciales (cf. section 7.4).

Il s'est avéré qu'il n'est pas facile de gérer l'interaction entre les amalgames, les composés, les formes détachables, les phénomènes de majuscule et les fautes d'orthographe, en particulier lorsque l'on traite le premier mot d'une phrase. Cependant, nous avons défini de nombreuses heuristiques qui donnent des résultats satisfaisants.

Globalement, TEXT2DAG fonctionne de la façon suivante:

– Une premier étape découpe et/ou rassemble les tokens pour en faire des *mots* (c'est-à-dire des formes simples ou des composants de formes composés, éventuellement à corriger). Pour cela, il se fonde sur une estimation du coût de la correction du token courant, du token précédent, et de leur concaténation, en simulant ces correction. Il essaye également de voir ce que devient la concaténation des tokens reconnus

comme une expression en langue étrangère, au cas où un espace surnuméraire en soit la cause (ainsi, `{disp aru}` _ETR doit devenir `{disp aru}` disparu). Un aperçu des heuristiques en jeu est donné en annexe (algorithmes 1 à 4: le premier précise la façon dont on essaye de « recoller » différents tokens en un seul; les trois suivants décrivent la façon dont les différents types de formes détachables sont gérés, en interaction avec la correction orthographique, simulée ou effective).

– Un deuxième étape se voit transmettre le flux de mots produits par l'étape précédente. Certains de ces mots sont déjà des formes correctes, d'autres sont à corriger, d'autres enfin sont des amalgames ou des composants de mots composés, voire plusieurs de ces choses à la fois. Il faut alors construire le DAG des formes, en prenant en compte tous ces phénomènes et en mettant en œuvre la correction ambiguë. L'heuristique sous-jacente est présentée en annexe (algorithme 5). On notera qu'un mot correct composant de forme composée, mais n'étant pas entouré par les autres composants de ce composé, doit être corrigé spécialement pour en faire une forme simple[34].

À l'exception du premier mot de la phrase, les mots inconnus à initiale majuscule pour lesquels la mise en minuscule et/ou l'ajout d'un diacritique sur certaines lettres conduit à une forme présente dans le lexique sont représentés comme une alternative entre le mot de départ et la forme modifiée présente dans le lexique. Enfin, il peut rester dans le DAG des mots inconnus, pour lesquels aucune correction n'a été trouvée dont le coût soit inférieur aux seuils définis par l'utilisateur. Ils sont laissés tels quels pour le moment. Ces formes inconnues constituant des transitions dans le DAG pourront être utilisées par les modules décrits à la section suivante[35], puis traités par le module `uw` (cf. section 7.4).

6.2.3. *Évaluation*

Nous avons procédé à une évaluation manuelle de TEXT2DAG sur des extraits des trois mêmes sous-corpus du corpus EASy qui nous ont servi à évaluer les grammaires locales de l'étape `txt2tok`. On peut constater à quel point la variété des genres induit des différences de fréquences entre les différents phénomènes. Naturellement, l'évaluation proposée ici dépend à la fois du jeu de règles utilisées, des paramètres[36] et du lexique. Le lexique utilisé, comme indiqué précédemment, est ici le Le*fff* (408 455 formes, 89 amalgames, 5 463 composés, 410 248 formes simples ou composants de composés).

---

34. Par exemple, `bric` n'existe que comme composant du multi-mot *de bric et de broc*. Ainsi, `un bric de glace` n'est corrigé qu'à la toute fin du processus en `un bris de glace`.
35. Ainsi, `Thabo Mbeki` n'étant pas une entrée du Le*fff*, il est souhaitable d'introduire dans le DAG une forme `Thabo` (connue) et une forme `Mbeki` (inconnue), qu'une grammaire de reconnaissance des noms propres, qui s'appuie éventuellement sur un lexique autre que le Le*fff*, pourra reconnaître comme désignant effectivement une personne.
36. Les paramètres utilisés sont ceux de la chaîne `sxpipe-easy`.

Corpus	general_elda	oral_elda_1	mail_1
changement de casse	21	2	17
changement de casse ambigu	18	12	2
du / des / de la	26	10	4
mots composés (ambigus)	12	17	17
au(x) / au(x)quel(s)	9	7	7
mots composés (non ambigus)	1	0	2
correction légère	3	1	3
correction lourde	1	0	19
correction lourde erronée	0	0	3
mots inconnus	1	15	1
détachement de préfixe	0	1	0
Nombre de tokens	*665*	*620*	*805*
Temps d'exécution	*0,23s*	*0,20s*	*0,28s*

Tableau 3. *Évaluation de* TEXT2DAG. *À l'exception des « corrections lourdes erronées », toutes les opérations sont correctes (ou introduisent un chemin correct parmi plusieurs, en cas d'ambiguïté).*

7. Enrichissement du DAG de formes

Une fois construit un DAG de formes, SXPipe 2 propose une architecture pour l'enrichissement et la modification de ce DAG, nommée `dag2dag`. Cette architecture, détaillée ci-dessous, repose sur un analyseur syntaxique non contextuel non déterministe à la Earley, qui utilise en entrée comme en sortie des DAG de formes. Chaque traitement repose sur une description non contextuelle de motifs à reconnaître, qui inclut des spécifications de transformations à appliquer au DAG en cas de reconnaissance de ces motifs. Ces actions peuvent être de trois types:

– insertion d'une forme spéciale (une balise ouvrante ou fermante),

– annotation d'une forme par une étiquette,

– remplacement d'un chemin entre deux nœuds du DAG par une forme spéciale (entité nommée).

Nous décrivons tout d'abord l'architecture `dag2dag`, avant d'en montrer l'efficacité par rapport à un moteur de gestion des expressions régulières souvent utilisé, celui du langage *perl*. Enfin, nous décrirons un exemple complet de grammaire `dag2dag` pour l'identification des occurrences impersonnelles du pronom *il*. Ce module, ILIMP 2, repose sur une grammaire éditée à l'aide de l'environnement d'édition de graphes d'Unitex, *via* un module de conversion nommé `unitex2sxpipe`.

7.1. *Reconnaissance de motifs non contextuels*

La reconnaissance de motifs dans des textes, parfois également dans des DAG de formes, est généralement réalisée au moyen d'expressions régulières, souvent en *perl*, ou d'automates finis (plus précisément de transducteurs). Nous avons choisi, au contraire, d'utiliser un analyseur syntaxique général qui sait traiter les grammaires non contextuelles, pour un certain nombre de raisons:

– la puissance d'expression des expressions régulières et des transducteurs (finis) n'est pas toujours suffisante (nous verrons ci-dessous que la grammaire ILIMP n'est pas une grammaire régulière),

– nous disposons, grâce au système SYNTAX (Boullier et Deschamp, 1988), d'un analyseur non contextuel très efficace qui garantit de bonnes performances bien que l'on dispose de toute la puissance des grammaires non contextuelles,

– l'analyseur à la Earley de SYNTAX gère l'ambiguïté au niveau de son entrée comme à celui de la grammaire utilisée: il peut prendre en entrée et produire en sortie des `dag` et des `udag`, aux formats décrits plus haut, en prenant en compte l'ambiguïté de la grammaire,

– l'analyseur à la Earley de SYNTAX est capable de gérer des grammaires de taille considérable (Boullier et Sagot, 2007), et des lexiques de plusieurs centaines de milliers de mots, voire plus, sans que l'efficacité n'en soit significativement dégradée,

– la manipulation de la forêt d'analyse non contextuelle permet le développement de filtres afin de ne conserver que les motifs recherchés (motifs de longueur maximale, motifs non consécutif, etc...).

Techniquement, les composants de `dag2dag` sont indépendants de la grammaire utilisée[37]. Chaque module utilisant l'architecture `dag2dag` doit définir un certain nombre de paramètres, une grammaire, ainsi qu'un *lexique* ou un analyseur lexical (selon le cas, cf. plus bas)[38].

7.1.1. *Construction de l'analyseur*

7.1.1.1. Analyseur à la Earley de SYNTAX

L'analyseur de `dag2dag` est l'analyseur à la Earley du système SYNTAX (Boullier et Deschamp, 1988)[39]. Il repose sur un automate coin-gauche sous-jacent et constitue une évolution de (Boullier, 2003). L'ensemble des analyses produites par cet analyseur est représenté par une forêt partagée. En réalité, cette forêt partagée peut être vue elle-même comme une grammaire non contextuelle dont les règles (ou productions) sont

37. Ils sont stockés dans le sous-dossier `dag2dag/common`.
38. Pour un module `dag2dag/module`, les paramètres forment les en-têtes du `Makefile.am` dudit module. La grammaire ainsi que le lexique ou l'analyseur lexical sont stockés dans `dag2dag/module/spec`.
39. SYNTAX est distribué librement, sous licence Cecill-C et Cecill selon les parties (licences compatibles respectivement avec LGPL et GPL). Site internet de SYNTAX: `http://syntax.gforge.inria.fr`.

des productions instanciées de la grammaire d'origine[40]. L'analyseur prend en entrée des DAG de formes. Savoir prendre des DAG en entrée ne nécessite pas de changements considérables dans l'algorithme d'Earley, au moins du point de vue théorique[41].

Depuis les travaux décrits dans (Boullier et Sagot, 2007), l'analyseur à la Earley de SYNTAX est à même de gérer des grammaires très volumineuses. L'activation de cette possibilité augmente légèrement les temps d'analyse mais permet la construction d'analyseurs impossibles à construire autrement, en raison de la taille excessive de la grammaire[42]. Cette activation se fait au moyen du paramètre `HUGE=1`.

7.1.1.2. Lexique et analyseur lexical

Le DAG de formes pris en entrée par l'analyseur est converti par le *lexeur* en un DAG de lexèmes (un lexème étant ici un symbole terminal de la grammaire CFG). Cette conversion peut se faire de deux façons:

– au moyen d'un analyseur lexical `lecl` (analyseur lexical du système SYNTAX): certaines formes sont des terminaux de la grammaire, d'autres sont identifiées par l'analyseur lexical, et les autres sont remplacées par un terminal par défaut, `__ANY__`;

– au moyen d'un lexique, c'est-à-dire ici un ensemble de couples *(forme,terminal)*: le lexique confère un certain nombre de terminaux à chaque forme qu'il contient, les autres formes se voyant attribuer un terminal par défaut, `__ANY__`

On notera que ces deux façons de faire ne sont pas exclusives l'une de l'autre: un module `dag2dag` peut définir à la fois un analyseur lexical et un lexique. Dans ce cas, l'éventuel résultat de l'analyse lexicale et les éventuels résultats donnés par le lexique sont accumulés. Ce n'est qu'en cas d'échec global que le terminal par défaut `__ANY__` est attribué[43].

---

40. Si A est un symbole non terminal de la grammaire G, $A_{i..j}$ est un *symbole non terminal instancié* si et seulement si $A \xRightarrow[G]{+} a_i \dots a_{j-1}$, où $w = a_1 \dots a_n$ est un chemin dans le DAG d'entrée et $\xRightarrow[G]{+}$ la fermeture transitive de la relation *dérive*.

41. Si i est un nœud du DAG et si l'on a une transition sur le terminal t vers le nœud j (sans perte de généralité, on peut supposer $j > i$) et si l'item Earley $[A \rightarrow \alpha \bullet t\beta, k]$ est un élément de la table $T[i]$, alors on peut ajouter à la table $T[j]$ l'item $[A \rightarrow \alpha t \bullet \beta, k]$ s'il n'y est pas déjà. Il faut faire attention à ne commencer une phase PREDICTOR dans une table $T[j]$ que si toutes les phases Earley (PREDICTOR, COMPLETOR et SCANNER) sont déjà achevées dans toutes les tables $T[i]$, $i < j$.

42. De façon informelle, l'activation du mécanisme de gestion de grosses grammaires est appropriée lorsque la taille de la grammaire (le nombre d'occurrences de symboles) dépasse environ 100 000.

43. L'activation d'un analyseur lexical se fait par le paramètre `LECL=1`, celle d'un lexique se fait par `LEXICON=1`.

7.1.1.3. Grammaire

Une grammaire `dag2dag` a vocation à définir des motifs[44]. Elle n'est pas à proprement parler une grammaire non contextuelle, en ceci qu'elle n'a pas d'axiome. En réalité, l'architecture `dag2dag` se charge de transformer une telle grammaire en une « vraie » grammaire non contextuelle, utilisée par l'analyseur, par l'introduction d'un axiome et d'un en-tête[45]. Cet en-tête permet la reconnaissance de zéro ou un motif dans chaque DAG d'entrée. C'est l'ambiguïté de l'analyseur qui permet de récupérer, avant filtrage, *tous* les motifs que l'on peut reconnaître dans le DAG d'entrée: outre l'analyse ne reconnaissant aucun motif, on disposera d'autant d'analyses qu'il y a de motifs différents. Le fait qu'ils soient imbriqués, croisés ou ambigus n'a donc aucun impact: ils sont tous reconnus indépendamment les uns des autres[46]. C'est sur cette base que se fait ensuite le filtrage et l'extraction du DAG de sortie (cf. ci-dessous).

Les règles de grammaires sont au format BNF de SYNTAX (les non-terminaux entre chevrons, les terminaux sans chevrons, éventuellement entre guillemets), illustré par l'exemple suivant:

```
<_NP!> = <ART?> <TITLE> <CAP> hyphen <CAP> <CAP*> ;
```

On note qu'il ne s'agit pas de BNF étendue: le point d'interrogation ou l'étoile à la fin des noms de symboles en font partie intégrante (ils n'ont pas leur signification fréquente d'opérateurs réguliers).

Toutefois, un non-terminal, comme `_NP!` se terminant par `!` indique à `dag2dag` qu'il faudra, au moment de la construction du DAG de sortie, effectuer une action particulière. Nous décrivons ceci en 7.1.3.

Après une règle (au-delà du point-virgule de fin), on peut associer un niveau de priorité à la règle[47]. Ce niveau de priorité sera exploité au moment du filtrage, comme expliqué ci-dessous.

7.1.2. *Construction et filtrage de la forêt*

L'analyseur syntaxique construit à partir d'une grammaire `dag2dag` construit, pour chaque DAG donné en entrée, une forêt d'analyse. Cette forêt représente d'une façon compacte et factorisée l'ensemble des analyses possibles, c'est-à-dire l'ensemble des motifs reconnaissables.

44. Symboles non-terminaux `PATTERN` et `MPATTERN`. La différence entre les deux est expliquée en 7.1.2.
45. L'en-tête dépend en réalité de la valeur de l'option `ESP`, décrite en 7.1.2.
46. Dans un outils comme Unitex, ce n'est pas le cas: un seul motif est reconnu dans chaque phrase, et la gestion des motifs imbriqués ou croisés (à recouvrement non nul) nécessite de recourir à des astuces, comme des exécutions multiples de l'outil.
47. Il doit s'agir d'un entier relatif. La priorité par défaut est 0. Une priorité plus élevée est favorisée par rapport à une priorité moins élevée.

À ce stade, un certain nombre de règles de filtrage peuvent être appliquées, en fonction des besoins, que nous détaillons brièvement dans leur ordre d'application.

filtrage de l'analyse hors-motifs (`FILTER=1`) : si au moins un motif est trouvé, la lecture hors-motifs est éliminée; cette option est généralement activée, mais elle peut être inappropriée pour une application comme la reconnaissance d'expressions figées (à côté d'une lecture figée, on peut souhaiter conserver la lecture de base);

maximisation de la longueur des motifs (`LM=1`) : si l'on reconnaît plusieurs motifs entre un même état e_n et différents états e_{m_1},...,e_{m_N}, alors seuls les motifs maximisant la longueur du chemin $e_n \cdots e_{m_i}$ sont conservés (c'est donc une version imparfaite d'un filtre qui conserverait les motifs les plus longs, en un sens qui reste à définir);

filtrage par priorités (dans la grammaire) : si un même non-terminal instancié est réécrit de différentes façons, au moyen de différentes règles, au sein de la forêt d'analyse, alors on élimine toutes les réécritures n'instanciant pas des règles de priorité optimale; si par exemple la forêt contient `<A`$_{i..j}$`> = <B`$_{i..j}$`> 1` ainsi que `<A`$_{i..j}$`> = <C`$_{i..k}$`> <D`$_{k..j}$`> 0`, alors seule la première lecture sera conservée, car elle fait usage d'une règle de priorité supérieure;

filtrage des sous-motifs (`ESP=1`) : si l'on reconnaît un motif entre l'était e_n et l'état e_m du DAG, et que l'on reconnaît également des motifs allant de e_n à e_{i_1}, de e_{i_1} à e_{i_2}, ..., et de e_{i_N} à e_m, alors on peut vouloir éliminer tous ces sous-motifs (c'est le cas par exemple pour une grammaire des nombres: si on reconnaît cent vingt comme un nombre, ainsi que cent et vingt, on ne veut conserver que la lecture où cent vingt est un nombre); ce filtre est le seul qui fasse la différence entre les `PATTERN`, seuls affectés, et les `MPATTERN`, non concernés;

7.1.3. *Construction du DAG de sortie*

Il reste maintenant à construire le DAG de sortie à partir de la forêt filtrée. On veut donc obtenir le DAG minimal regroupant l'union de tous les chemins correspondant à chaque analyse. C'est cette union qui permettra de représenter les ambiguïtés (et donc l'existence possible de plusieurs motifs dans une phrase, y compris dans un même chemin).

Cette opération est réalisée au moyen d'un parcours descendant récursif de la forêt filtrée, de la façon suivante. Lorsque le parcours nous amène sur un nœud *ET*, on rend le sous-DAG obtenu par concaténation des sous-DAG des fils. Lorsque le parcours nous amène sur un nœud *OU*, on rend le sous-DAG obtenu par disjonction entre les sous-DAG des fils. Lorsque l'on arrive sur une feuille, on sort la forme associée à cette feuille, précédée de son « commentaire » (cf. plus haut). Il y a toutefois un certain nombre de nœuds particuliers, qui permettent d'annoter le DAG produit de différentes façons:

forme annotée : lorsque l'on arrive sur une feuille dont le terminal[48] se termine par le caractère `!`, alors ce terminal (sans ledit caractère) est utilisé pour étiqueter la forme; exemple: la règle `<PATTERN> = ilimp! pleut ;` appliquée (sans lexique) au DAG linéaire il pleut produira en sortie il__ilimp pleut;

balises : lorsque l'on arrive sur un non-terminal (nœud *OU*) dont le nom se termine par `!` et commence par `:`, on est en présence d'une balise ouvrante: ce non-terminal, qui est censé se réécrire en la chaîne vide , est sorti(sans son `!`) comme si c'était un terminal, associé au commentaire de la forme qui le *suit*[49]; un non-terminal se terminant par `:!` est une balise fermante: tout fonctionne de la même façon, sauf que le commentaire associé est celui de la forme qui précède[50]; exemple: la règle `<PATTERN> = <TIME:!> %int <MONTH> <:TIME!>`, appliquée à 21 décembre, produira en sortie `TIME: 21 décembre :TIME`;

entité nommée : lorsque l'on arrive sur un non-terminal (nœud *OU*) dont le nom se termine par `!` sans qu'il s'agisse d'une balise, la sous-forêt dont ce non-terminal est la racine est remplacée par la forme spéciale obtenue en enlevant le `!` final (le commentaire associé à cette forme spéciale est obtenue en rassemblant tous les commentaires des formes qu'elle subsume); exemple: la règle `<_NP!> = <ART?> <TITLE> <CAP> hyphen <CAP> <CAP*> ;`, en supposant qu'elle reconnaisse Dr. Jean - Pierre Dupont au sein de la phrase ceci est pour le Dr. Jean - Pierre Dupont, induira la production en sortie de ceci est pour `{Dr. Jean - Pierre Dupont}_NP`;

Une fois une première version du DAG de sortie obtenue de cette façon, un algorithme de minimisation est appliquée. C'est son résultat qui est effectivement sorti.

7.2. *Évaluation quantitative: l'exemple de la grammaire des nombres*

Nous allons illustrer l'efficacité de l'architecture `dag2dag`, et en particulier celle des analyseurs à la Earley construits par SYNTAX, sur un exemple précis: une grammaire des nombres et autres expressions utilisant des éléments numériques, appelée `numbers`. En effet, dans une version antérieure de SXPipe, une séquence (linéaire) de mots (formes simples ou composants de mots composés) était transmise successivement à deux grammaires, destinées à y reconnaître les expressions à contenu numérique (nombres écrits en toutes lettres, dates non reconnues précédemment, par exemple à cause d'une faute d'orthographe, etc.). Ces grammaires étaient alors des cascades d'expressions régulières *perl*, aux propriétés suivantes:

– entrée linéaire

– déterministe

– sortie linéaire

48. Qu'il ait été obtenu *via* un lexique ou par l'analyseur lexical
49. Dans le cas d'un DAG, ladite forme n'est pas nécessairement définie de manière unique. On en choisit une arbitrairement.
50. Même remarque.

Nous avons converti ces grammaires en une unique grammaire pour `dag2dag`. Naturellement, la puissance complète des grammaires non contextuelles est ici inutile. Mais `dag2dag` permet de conférer au module `numbers` les propriétés suivantes:

– DAG en entrée

– non-déterministe

– DAG en sortie

Pour autant, l'efficacité du module n'en pâtit pas, bien au contraire, comme le montre l'expérience suivante. Nous avons tout d'abord utilisé une version antérieure de SXPipe pour construire, à partir d'un corpus journalistique de 2697 phrases[51], une séquence (linéaire) de mots. Le résultat découpe ces phrases en 86 590 mots au total. Nous avons alors donné cette séquence aux deux modules faisant usage de notre grammaire des nombres: le module reposant sur les deux cascades d'expressions régulières *perl* et le module `numbers` mettant en œuvre l'architecture `dag2dag`.

Le résultat est édifiant: les expressions régulières *perl* mettent 1,72 secondes là où `numbers` (le module SYNTAX/`dag2dag`) met 0,55 secondes, alors que ce dernier effectue un travail de bien meilleure qualité, puisque les éventuelles ambiguïtés sont correctement traitées.

Ceci montre que l'utilisation de `dag2dag`, malgré sa puissance supérieure à celle des expressions régulières, ne porte pas atteinte à l'efficacité du module, bien au contraire.

7.3. *Intégration de* ILIMP *dans* SX*Pipe*

Ainsi qu'évoqué à la section 2, le système Unitex permet, entre autres, de reconnaître automatiquement des motifs dans un texte brut. Grâce à son interface graphique, il permet aux linguistes de construire des grammaires de façon quasi intuitive, sous forme de réseaux de transitions récursifs, équivalents à des grammaires non contextuelles (CFG). Il souffre toutefois de limitations importantes, dont l'absence de gestion des ambiguïtés, l'absence de gestion satisfaisante de cascades de traitement, l'impossibilité de prioriser l'application d'une règle par rapport à une autre, et des problèmes d'efficacité.

À l'inverse, SXPipe, grâce à `dag2dag`, est à même de traiter de façon satisfaisante des grammaires non contextuelles. C'est la confrontation de ces deux constats qui a motivé le développement d'un outil, `unitex2sxpipe`, pour permettre la conversion d'une grammaire Unitex (un ensemble de « graphes » et éventuellement un lexique associé) en une grammaire `dag2dag` (et éventuellement un lexique associé), construisant ainsi un outil nommé ILIMP 2. Cet outil a été appliqué à différentes grammaires Unitex, dont ILIMP, développée par Laurence Danlos (Danlos, 2005), qui distingue

51. Il s'agit du sous-corpus dit `general_lemonde` du corpus utilisé pour la campagne EASy d'évaluation des analyseurs syntaxiques pour le français.

les occurrences anaphoriques et impersonnelles du pronom « *il* », et obtient 97,5% de précision sur un corpus journalistique de référence.

Ici encore, les résultats sont probants. D'une part, il n'est plus nécessaire d'appliquer ILIMP trois fois de suite pour obtenir approximativement les annotations recherchées: la gestion complète des ambiguïtés par `dag2dag`, ainsi que la possibilité de donner des priorités aux règles, permettent de ne réaliser qu'un traitement unique. De plus, ILIMP 2, en tant que composant de SXPipe, bénéficie de tous les traitements qui le précèdent, dont la correction orthographique et la reconnaissance de mots composés. Enfin, l'efficacité du traitement est améliorée: sur *le tour du monde en 80 jours* (68 000 mots, 1 500 occurrences de il), la première des trois passes nécessaires sous Unitex prend 1 minute 30, alors que le traitement *complet* par ILIMP 2 (sous SXPipe) prend 48 secondes. A contrario, SXPipe bénéficie ainsi de l'environnement graphique de développement de grammaires sous forme de réseaux de transitions récursifs.

D'autres grammaires Unitex ont ainsi pu être converties en un module `dag2dag`. C'est le cas de la grammaire *time_french*, développée par Maurice Gross au LADL et distribuée par l'IGM, qui reconnaît les expressions temporelles.

7.4. *Gestion des mots inconnus non corrigés*

Une fois tous les traitements effectués, il peut rester dans le DAG de sortie des formes qui sont inconnues du lexique. Pour elles, aucune correction n'a été trouvée dont le coût soit inférieur aux seuils définis par l'utilisateur. Le module `dag2dag` de gestion des mots inconnus, `uw`, peut alors les remplacer par l'une ou l'autre des deux formes spéciales _uw et _Uw, présentes dans le lexique. La forme _uw représente les mots inconnus entièrement en minuscule (le L*efff* considère _uw comme un mot pouvant avoir toutes les catégories ouvertes), alors que _Uw représente les mots inconnus comportant une majuscule (le L*efff* considère _Uw comme pouvant être un nom propre ou un nom commun). Lorsqu'il s'agit du premier mot d'une phrase, et que le mot inconnu comporte une majuscule, les deux semblent possible. Dans ce cas, le module `uw` le remplace par l'alternative (_uw | _Uw).

Nous envisageons de proposer, à côté de ce module simple, un module plus complexe d'analyse morphologique des mots inconnus. Ce module transformerait alors certains mots inconnus en des formes spéciales plus spécifiées que _uw et _Uw. On peut par exemple imaginer que les formes conjuguées de verbes inconnus en -iser ou -ifier soient remplacées par des formes spéciales connues du lexique que l'on pourrait nommer _uw_v_-ifie/_uw_v_-ise, _uw_v_-ifions/_uw_v_-isons, etc. Toutes ces formes seraient des entrées (spéciales) du L*efff*, correspondant aux « formes fléchies » du « lemme » _uw_v_-ifier.

8. Résultats quantitatifs sur un gros corpus

Faute de corpus de référence pour la plupart des traitements appliqués par SXPipe, nous ne sommes pas encore en mesure de proposer une analyse qualitative complète de la chaîne. À la section suivante, nous indiquons toutefois des pistes qui devraient nous permettre d'avoir une bonne idée de la qualité du résultat de l'ensemble de la chaîne SXPipe sur différents corpus (dans sa configuration par défaut), et en particulier de son impact sur la qualité des résultats d'analyseurs syntaxiques utilisés en aval.

Nous avons cependant effectué une évaluation quantitative de la chaîne (temps d'exécution, caractéristiques des résultats) sur un corpus journalistique volumineux issu du *Monde diplomatique*. Ce corpus fait presque 100 millions de caractères, dont 81 millions de caractères pleins (ni espaces ni sauts de lignes).

La phase 2 de SXPipe découpe ce corpus en 962 000 phrases environ, constituées de 17,5 millions de tokens. À l'issue de l'application de la chaîne complète, les 962 000 DAG construits rassemblent 21 millions de formes. Le nombre de formes (c'est-à-dire de transitions) dans le DAG associé à une phrase est donc de l'ordre de 1,2 fois le nombre de tokens qui la constituent.

Le tableau 4 récapitule les 5 cas d'ambiguïté les plus fréquents, ainsi que leur nombre d'occurrences dans le corpus, ainsi que quelques autres cas fréquents et intéressants. On notera que la forme spéciale _EPSILON, que nous n'avons pas mentionné jusqu'à présent, correspond sans surprise à une ε-transition dans le DAG, c'est-à-dire une transition que les analyseurs syntaxiques (ou tout autre outil venant en aval) devra « sauter ». Ainsi, un DAG comme (" | _EPSILON) signifie que le guillemet peut être soit reconnu comme une forme soit ignoré. Il en va de même pour les autres formes d'ambiguïté: les analyseurs syntaxiques (éventuellement après une phase d'étiquetage morphosyntaxique), pourront choisir, dans chaque cas d'ambiguïté dans le DAG, le ou les chemins pour lesquels une analyse syntaxique est possible.

Le temps d'exécution de différentes étapes (cf. section 4) ou modules est indiqué à la table 5, ainsi que le temps total: les 100 millions de caractères ont été traités complètement par SXPipe en 15 234 secondes (environ 4 heures et 14 minutes)[52]. Le module la plus coûteuse en temps de calcul, et de loin, est TEXT2DAG.

9. Conclusion et perspectives

Nous avons présenté SXPipe 2, une chaîne de pré-traitements pour le français. SXPipe procède en différentes étapes paramétrables et modulaires qui permettent de convertir efficacement un texte brut en un DAG de formes. Ceci est effectué au moyen de grammaires locales, dont des descriptions de motifs non contextuels permettant une reconnaissance non-déterministe, d'outils de segmentation et de tokenisation et

52. Ces temps d'exécution ont été obtenus sur une machine disposant d'un processeur à 2 cœurs (Intel Core 2 Duo à 2,4 GHz) et de 2 Go de mémoire vive.

Rang	Token(s) de départ	DAG produit	occ.
1	"	(" | _EPSILON)	295 326
2	des	(des | de$_{\text{prep}}$ les$_{\text{det}}$)	279 453
3	du	(du | de$_{\text{prep}}$ le$_{\text{det}}$)	172 384
4	de la	(de_la | de$_{\text{prep}}$ la$_{\text{det}}$)	127 201
5	Les	(Les | les)	35 198
8	A	(a | à)	11 870
15	plus de	(plus_de | plus de)	5 076
23	alors que	(alors_que | alors que)	3 694
27	de plus en plus	(de_plus_en_plus | (de_plus | de plus) (en_plus | en plus))	3 576
45	d'ailleurs	(d'ailleurs | d' ailleurs)	2 867
47	d'abord	(d'abord | d' abord)	2 862
49	il y a	(il_y_a | il y a)	2 834
62	grâce à	(grâce_à | grâce à)	2 018

Tableau 4. *Ambiguïtés dans les DAG produits par* SX*Pipe sur un corpus journalistique de 17,5 millions de tokens. Les 5 ambiguïtés les plus fréquentes, ainsi que quelques autres, ont été indiquées.*

Phase / Module(s)	Temps d'exécution (s) / nombre de cœurs exploités
1 (txt2tok) / *tous les modules*	2 643 / 2
2 (tokeniseur/segmenteur)	222 / 1
3 (tok2dag) / *tous les modules*	1 200 / 2
4 (TEXT2DAG)	14 828 / 1
5 (dag2dag) / numbers	783 / 1
5 (dag2dag) / np	539 / 1
5 (dag2dag) / uw	238 / 1
Chaîne complète	**15 234** / 1

Tableau 5. *Temps d'exécution de différentes étapes (cf. section 4) et modules de* SX*Pipe sur un corpus journalistique de 17,5 millions de tokens. Les modules* dag2dag *nommés* numbers, np *et* uw *reconnaissent respectivement les nombes (cf. section 7.2), les noms propres et les formes inconnues restant dans les DAG (cf. section 7.4). On notera que (pour le moment) les modules des étapes 1 et 2, ainsi que les grammaires locales de l'étape 3, sont écrites en* perl*. Les modules des étapes 4 et 5 sont écrits en C (et reposent sur tout ou partie du système* SYNTAX*), et sont donc plus bien efficaces, compte tenu du travail qu'ils effectuent. Il en va de même du premier module de l'étape 3, qui identifie les tokens dits* inanalysables*. Le total est approximativement égal au temps d'exécution de* TEXT2DAG*: de façon imagée, un des deux cœurs exécute* TEXT2DAG *pendant que l'autre, sans être utilisé à 100%, exécute l'ensemble des autres modules.*

d'un module non-déterministe de correction orthographique et de reconnaissance de formes composées. L'ensemble permet la construction de chaînes efficaces. La chaîne par défaut pour le français obtient des résultats très satisfaisants en termes de précision et de rappel dans les différentes tâches qu'elle réalise.

Si SXPipe 2 met particulièrement l'accent sur la gestion des ambiguïtés, les travaux à venir se concentreront sur l'adaptabilité dynamique de la chaîne, dans un contexte multilingue et multi-genres. L'idée est qu'une première passe sur un document extraira des informations (langue, genre, noms propres, etc.), lesquelles permettront une adaptation à la volée des modules en vue du traitement proprement dit (changement de lexique, paramétrisation des grammaires en fonction de la langue, activation ou désactivation de modules, changement des paramètres du correcteur orthographique). Par ailleurs, une gestion plus avancée des dépendances entre modules permettra de faciliter la définition de chaînes particulières destinées à une tâche donnée.

L'ensemble de ces avancées permettra d'améliorer encore la qualité des prétraitements effectués par SXPipe, et donc la robustesse et la précision de tout outil utilisé en aval, qu'il s'agisse d'un analyseur syntaxique profond, d'un système d'extraction ou de recherche d'informations, d'un outil de normalisation textuelle, ou de tout autre traitement sur corpus.

C'est grâce à de telles applications, et notamment l'analyse syntaxique, qu'une évaluation qualitative globale satisfaisante de SXPipe pourra être effectuée. Nous envisageons par exemple d'appliquer SXPipe à un corpus arboré avant l'entraînement d'un analyseur probabiliste, et de comparer le résultat à l'analyseur entraîné sur le corpus initial. De plus, ces deux analyseurs pourraient être évalués qualitativement, par exemple sur le corpus EASy, qui inclut des sous-corpus de genres très différents, y compris des genres pour lesquels l'utilisation de SXPipe s'est déjà avérée indispensable (transcription de corpus oraux, corpus de courriers électroniques...). Nous espérons montrer par ces expériences l'utilité du pré-traitement et la qualité de SXPipe pour effectuer cette tâche.

10. Bibliographie

Bilhaut F., Widlöcher A., « LinguaStream: An Integrated Environment for Computational Linguistics Experimentation », *Proceedings of the 11th Conference of the European Chapter of the Association of Computational Linguistics (EACL) (Companion Volume)*, Trento, Italie, p. 95-98, avril, 2006.

Boullier P., « Guided Earley Parsing », *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT'03)*, Nancy, France, p. 43-54, 2003.

Boullier P., Clément L., Sagot B., Villemonte de la Clergerie E., « Chaînes de traitement syntaxique », *Actes de TALN'05*, ATALA, Dourdan, France, p. 103-112, June, 2005.

Boullier P., Deschamp P., « Le système SYNTAX™ – Manuel d'utilisation et de mise en œuvre sous UNIX™ », , En ligne sur `http://syntax.gforge.inria.fr/syntax3.8-manual.pdf`, 1988.

Boullier P., Sagot B., « Are very large context-free grammar tractable? », *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT'07)*, Prague, République Tchèque, 2007.

Clément L., de La Clergerie E., « Terminology and other language resources – Morpho-Syntactic Annotation Framework (MAF) », 2004, ISO TC37SC4 WG2 Working Draft.

Clément L., Villemonte de La Clergerie E., « MAF: a Morphosyntactic Annotation Framework », *Proceedings of the 2nd Language & Technology Conference (LT'05)*, Poznań, Poland, p. 90-94, April, 2005.

Cunningham H., Maynard D., Bontcheva K., Tablan V., « GATE: A framework and graphical development environment for robust NLP tools and applications », *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics*, 2002.

Danlos L., « ILIMP : Outil pour repérer les occurences du pronom impersonnel *il* », *Actes de TALN 2005*, Dourdan, France, 2005.

Grefenstette G., Tapanainen P., « What is a Word, What is a Sentence? Problems of Tokenization », *Proceedings of the 3rd Conference on Computational Lexicography and Text Research*, Budapest, Hungary, 1994.

Heitz T., « Modélisation du prétraitement des textes », *Proceedings of JADT'06 (International Conference on Statistical Analysis of Textual Data)*, vol. 1, p. 499-506, 2006.

Kukich K., « Techniques for Automatically Correcting Words in Text », *ACM Computing Surveys*, vol. 24, n° 4, p. 377-439, dec, 1992.

Maynard D., Tablan V., Ursu C., Cunningham H., Wilks Y., « Named Entity Recognition from Diverse Text Types », *Proceedings of RANLP 2001*, Tzigov Chark, Bulgaria, 2001.

Oflazer K., « Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction », *Comp. Ling.*, vol. 22, n° 1, p. 73-89, 1996.

Paumier S., De la reconnaissance de formes linguistiques à l'analyse syntaxique, Thèse de doctorat, Université de Marne-la-Vallée, 2003. Jury : Ch. Choffrut, F. Guenthner, É. Laporte, J.S. Nam and D. Perrin.

Sagot B., Boullier P., « From raw corpus to word lattices: robust pre-parsing processing with SXPipe », *Archives of Control Sciences, special issue on Language and Technology*, vol. 15, n° 4, p. 653-662, 2005.

Sagot B., Clément L., Villemonte de La Clergerie E., Boullier P., « The L*efff* 2 syntactic lexicon for French: architecture, acquisition, use », *Proc. of LREC'06*, 2006.

Sagot B., Danlos L., Désir A., « Traitements de surface à l'aide de graphes de transitions récursifs: de Unitex à SXPipe », *Actes de TALN'08*, Avignon, France, 2008. soumis.

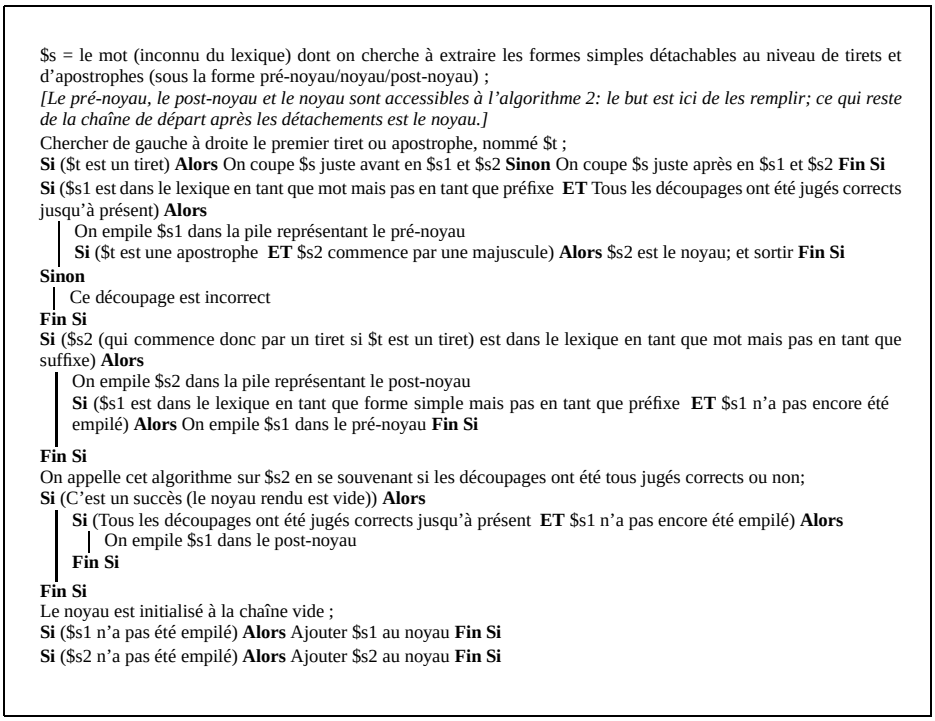
Annexe: heuristiques pour la correction orthographique et l'identification de formes composées

[Dans cet algorithme, « correction » signifie « correction globale » telle que décrite dans l'algorithme 2]
Tant que (Le token courant est non vide) **faire**
  Si (on cherche à corriger _ETR **ET** son commentaire associé est non vide) **Alors**
    Essayer de corriger la concaténation des tokens originels;
    Si (C'est un succès) **Alors** Transmettre la séquence de mots produite (0 correction en attente) **Fin Si**
  Fin Si
Fait
Essayer de corriger le token courant;
Si (on n'a pas de correction en attente **OU** le token précédent s'est corrigé à coût nul **ET** le token courant aussi)
Alors
  Transmettre la séquence de mots produite par la correction du token précédent, mettre en attente la correction du token courant et passer au token suivant;
Sinon
  Essayer de corriger la concaténation du token précédent et du token courant
  Si (Échec **OU** Succès de coût supérieur à la somme des coûts des corrections des 2 tokens) **Alors**
    Transmettre la séquence de mots produite par la correction du token précédent, mettre en attente la correction du token courant et passer au token suivant
  Sinon
    Transmettre la séquence de mots produite par la correction de la concaténation des deux tokens (plus de correction en attente)
  Fin Si
Fin Si
Transmettre la séquence de mots produite par la correction en attente, s'il y en a une

Algorithme 1: Heuristique simplifiée pour TEXT2DAG: manipulation des tokens.

Si ($s, la chaîne à corriger, est dans le lexique) **Alors** Rendre $s et sortir **Fin Si**
Si ($s peut être corrigé par correction « légère » **ET** $s n'a pas de majuscule) **Alors** Rendre $s et sortir **Fin Si**
Si (l'équivalent en minuscules de $s, est dans le lexique) **Alors** Rendre $s et sortir **Fin Si**
Si ($s est le premier mot de la phrase **ET** $n commence par une majuscule) **Alors**
  Découper l'équivalent en minuscules de $s en pré-noyau/noyau/post-noyau (voir algorithme 3)
Sinon
  Découper $s en pré-noyau/noyau/post-noyau
Fin Si
Si (Le noyau est vide (succès)) **Alors**
  Rendre le pré-noyau et le post-noyau, et sortir
Sinon
  Si ($n commence au début de la phrase **ET** $n commence par une majuscule) **Alors**
    Découper l'équivalent en minuscules de $n en préfixes/cœur/suffixes (voir algorithme 4)
  Sinon
    Découper $n en préfixes/cœur/suffixes
  Fin Si
  Si (Le cœur obtenu est dans le lexique) **Alors**
    Rendre le pré-noyau, les préfixes, le cœur, les suffixes et le post-noyau, et sortir
  Fin Si
  Si ($n a au moins une majuscule) **Alors** Rendre le pré-noyau, $n et le post-noyau, et sortir **Fin Si**
  Se souvenir du coût d'une correction standard de $n, ou l'impossibilité de le corriger ;
  [Ce coût est utilisé par l'algorithme 1]
  Rendre le pré-noyau, cette correction et le post-noyau, et sortir
Fin Si

Algorithme 2: Heuristique simplifiée pour TEXT2DAG: gestion des détachements.

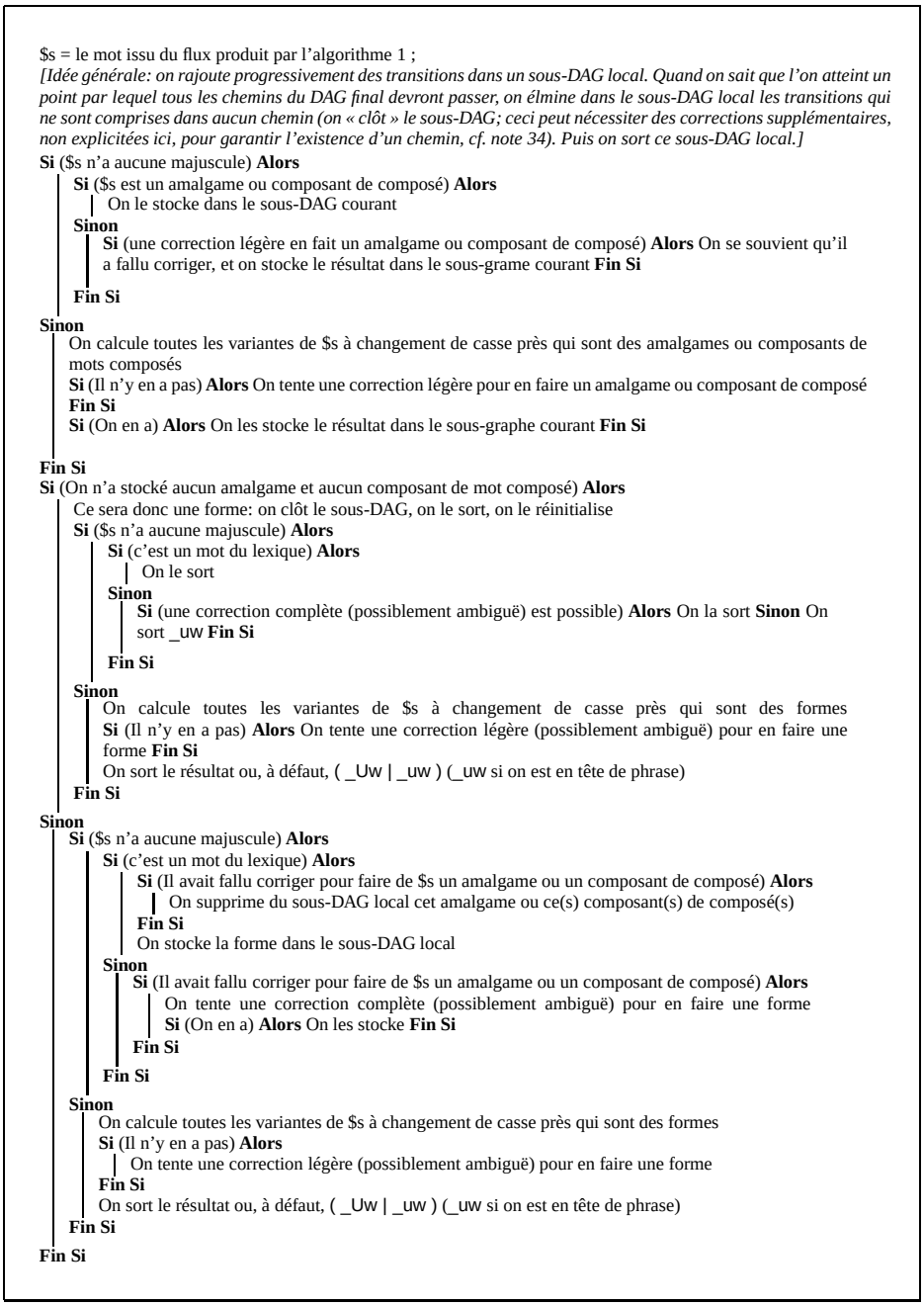


$s = le mot (inconnu du lexique) dont on cherche à extraire les formes simples détachables au niveau de tirets et d'apostrophes (sous la forme pré-noyau/noyau/post-noyau) ;
[Le pré-noyau, le post-noyau et le noyau sont accessibles à l'algorithme 2: le but est ici de les remplir; ce qui reste de la chaîne de départ après les détachements est le noyau.]
Chercher de gauche à droite le premier tiret ou apostrophe, nommé $t ;
Si ($t est un tiret) **Alors** On coupe $s juste avant en $s1 et $s2 **Sinon** On coupe $s juste après en $s1 et $s2 **Fin Si**
Si ($s1 est dans le lexique en tant que mot mais pas en tant que préfixe **ET** Tous les découpages ont été jugés corrects jusqu'à présent) **Alors**
  On empile $s1 dans la pile représentant le pré-noyau
  Si ($t est une apostrophe **ET** $s2 commence par une majuscule) **Alors** $s2 est le noyau; et sortir **Fin Si**
Sinon
  Ce découpage est incorrect
Fin Si
Si ($s2 (qui commence donc par un tiret si $t est un tiret) est dans le lexique en tant que mot mais pas en tant que suffixe) **Alors**
  On empile $s2 dans la pile représentant le post-noyau
  Si ($s1 est dans le lexique en tant que forme simple mais pas en tant que préfixe **ET** $s1 n'a pas encore été empilé) **Alors** On empile $s1 dans le pré-noyau **Fin Si**
Fin Si
On appelle cet algorithme sur $s2 en se souvenant si les découpages ont été tous jugés corrects ou non;
Si (C'est un succès (le noyau rendu est vide)) **Alors**
  Si (Tous les découpages ont été jugés corrects jusqu'à présent **ET** $s1 n'a pas encore été empilé) **Alors**
    On empile $s1 dans le post-noyau
  Fin Si
Fin Si
Le noyau est initialisé à la chaîne vide ;
Si ($s1 n'a pas été empilé) **Alors** Ajouter $s1 au noyau **Fin Si**
Si ($s2 n'a pas été empilé) **Alors** Ajouter $s2 au noyau **Fin Si**

Algorithme 3: Heuristique simplifiée pour TEXT2DAG: formes simples séparées par des tirets ou des apostrophes.

```
$s = le mot (inconnu du lexique) dont on cherche à extraire les préfixes et les suffixes détachables (sous la forme préfixes/cœur/suffixes) ;
[les préfixes, les suffixes et le cœur sont accessibles à l'algorithme 2: le but est ici de les remplir; ce qui reste de la chaîne de départ après les détachements d'affixes est le cœur.]
Tant que ($s se termine par un suffixe présent dans le lexique des suffixes) faire
    Retirer ce suffixe de $s
    Effectuer sur ce suffixe l'opération qui lui est associée (typiquement, le faire précéder de « _ »)
    BOUCLE: Tant que ($s est non vide) faire
        Si ($s est dans le lexique) Alors Sortir Fin Si
        Pour i de 1 à 1 + Nombre de tirets dans $s faire
            Noter $p ce qui est à gauche du i-ième tiret le plus à droite
            (tout $s si$i = 1 + Nombre de tirets dans $s);
            Si ($s ne commence pas par un préfixe ET $p ne comporte pas de tiret ni de majuscule
            ET $s n'est pas en début de phrase) Alors
                Essayer une correction « légère » de $p ;
                Si (Cette correction à marché) Alors Sortir de BOUCLE Fin Si
            Fin Si
        Fin Pour
    Fait
    Si ($p (éventuellement corrigé) est un préfixe) Alors Sortir Fin Si
    Essayer d'effectuer sur le préfixe $p l'opération qui lui est associée (rien, lui ajouter un tiret, ou autre) ;
    [Cela peut échouer, par exemple si on essaye de détacher un préfixe comme im-, qui spécifie qu'il doit précéder un m, un b ou un p, d'un radical qui commence par une autre lettre]
    Si (C'est un échec OU le reste du mot commence par une majuscule OU $p-tiret-reste du mot est un mot du lexique) Alors Sortir Fin Si
    Empiler $p dans la pile des préfixes
Fait
```

Algorithme 4: Heuristique simplifiée pour TEXT2DAG: identification des affixes.



```
$s = le mot issu du flux produit par l'algorithme 1 ;
[Idée générale: on rajoute progressivement des transitions dans un sous-DAG local. Quand on sait que l'on atteint un
point par lequel tous les chemins du DAG final devront passer, on élmine dans le sous-DAG local les transitions qui
ne sont comprises dans aucun chemin (on « clôt » le sous-DAG; ceci peut nécessiter des corrections supplémentaires,
non explicitées ici, pour garantir l'existence d'un chemin, cf. note 34). Puis on sort ce sous-DAG local.]
Si ($s n'a aucune majuscule) Alors
    Si ($s est un amalgame ou composant de composé) Alors
        On le stocke dans le sous-DAG courant
    Sinon
        Si (une correction légère en fait un amalgame ou composant de composé) Alors On se souvient qu'il
        a fallu corriger, et on stocke le résultat dans le sous-grame courant Fin Si
    Fin Si
Sinon
    On calcule toutes les variantes de $s à changement de casse près qui sont des amalgames ou composants de
    mots composés
    Si (Il n'y en a pas) Alors On tente une correction légère pour en faire un amalgame ou composant de composé
    Fin Si
    Si (On en a) Alors On les stocke le résultat dans le sous-graphe courant Fin Si
Fin Si
Si (On n'a stocké aucun amalgame et aucun composant de mot composé) Alors
    Ce sera donc une forme: on clôt le sous-DAG, on le sort, on le réinitialise
    Si ($s n'a aucune majuscule) Alors
        Si (c'est un mot du lexique) Alors
            On le sort
        Sinon
            Si (une correction complète (possiblement ambiguë) est possible) Alors On la sort Sinon On
            sort _uw Fin Si
        Fin Si
    Sinon
        On calcule toutes les variantes de $s à changement de casse près qui sont des formes
        Si (Il n'y en a pas) Alors On tente une correction légère (possiblement ambiguë) pour en faire une
        forme Fin Si
        On sort le résultat ou, à défaut, ( _Uw | _uw ) (_uw si on est en tête de phrase)
    Fin Si
Sinon
    Si ($s n'a aucune majuscule) Alors
        Si (c'est un mot du lexique) Alors
            Si (Il avait fallu corriger pour faire de $s un amalgame ou un composant de composé) Alors
                On supprime du sous-DAG local cet amalgame ou ce(s) composant(s) de composé(s)
            Fin Si
            On stocke la forme dans le sous-DAG local
        Sinon
            Si (Il avait fallu corriger pour faire de $s un amalgame ou un composant de composé) Alors
                On tente une correction complète (possiblement ambiguë) pour en faire une forme
                Si (On en a) Alors On les stocke Fin Si
            Fin Si
        Fin Si
    Sinon
        On calcule toutes les variantes de $s à changement de casse près qui sont des formes
        Si (Il n'y en a pas) Alors
            On tente une correction légère (possiblement ambiguë) pour en faire une forme
        Fin Si
        On sort le résultat ou, à défaut, ( _Uw | _uw ) (_uw si on est en tête de phrase)
    Fin Si
Fin Si
```

Algorithme 5: Heuristique simplifiée pour TEXT2DAG: correction ambiguë et gestion des composés.